Contrer l'attaque Simple Power Analysis efficacement dans les applications de la cryptographie asymétrique, algorithmes et implantations

Jean-Marc Robert

Collège Doctoral du Languedoc-Roussillon

THÈSE

Pour obtenir le grade de

Docteur

Délivré par l'**UNIVERSITÉ DE PERPIGNAN VIA DOMITIA**

Préparée au sein de l'école doctorale **ED305**
Et de l'unité de recherche **DALI/LIRMM**

Spécialité: **Informatique**

Présentée par **Jean-Marc ROBERT**

Contrer l'attaque *Simple Power Analysis* efficacement dans les applications de la cryptographie asymétrique, algorithmes et implantations

Soutenue le 08/12/2015 devant le jury composé de

M. Jean-Claude BAJARD	Pr.	UPMC	Examinateur
M. Thomas PLANTARD	CR	UOW	Examinateur
M. Christophe CLAVIER	Pr.	ULimoges	Rapporteur
M. Arnaud TISSERAND	DR	CNRS	Rapporteur
M. Bernard GOOSSENS	Pr.	UPVD	Directeur de thèse
M. Christophe NÈGRE	MC	UPVD	Co-directeur de thèse

UNIVERSITÉ PERPIGNAN VIA DOMITIA

Remerciements

Préparer cette thèse, la rédiger et la soutenir, ont été des privilèges pour lesquels je souhaite exprimer ma gratitude et ma reconnaissance. Elles s'expriment envers toutes les personnes qui m'ont soutenu, aidé et encouragé durant ces trois années.

En premier lieu, je remercie mon directeur de thèse, Bernard Goossens, sans qui rien n'aurait été possible, et qui m'a encouragé par ses conseils.

Je remercie vivement Christophe Nègre, qui m'a encadré au quotidien et qui a aussi été l'initiateur de cette thèse. Il m'a aussi aiguillé vers des voies de recherches et, tout au long de ces trois années, n'a jamais cessé de me conseiller et diriger pour que ce travail soit de la plus grande qualité.

Cette thèse a été financée par le projet ANR PAVOIS. Je suis donc redevable à l'ANR, et surtout à Arnaud Tisserand, qui a été le porteur de ce projet, et qui a également toujours été un encouragement et un soutien que j'ai particulièrement apprécié.

Durant ces trois années, j'ai aussi eu l'opportunité d'effectuer un séjour de dix mois à l'université de Wollongong, en Australie, sous la supervision de Willy Susilo et Thomas Plantard, envers lesquels va ma gratitude. Je remercie tout particulièrement Thomas Plantard, qui a dirigé mes travaux à Wollongong et dont l'inventivité a été une source d'inspiration d'une grande richesse.

Je remercie Christophe Clavier et Arnaud Tisserand à nouveau, qui en tant que rapporteurs, m'ont donné des conseils précieux pour faire de ce manuscrit un document de la meilleure qualité. Je suis aussi reconnaissant pour les encouragements qu'ils ont formulés dans leurs rapports.

Ces trois années, je les ai aussi passées au sein de l'équipe DALI/LIRMM, à l'Université de Perpignan Via Domitia. Je remercie l'ensemble de l'équipe, l'école doctorale ED305 et au delà toutes les personnes de l'Université de Perpignan qui ont été disponibles pour m'aider et m'encourager.

Je remercie Christian Berbain, relecteur bénévole, qui a grandement dégrossi la traque des diverses fautes de frappe, d'orthographe et de grammaire.

Enfin, en dernier lieu (*last but not least !*), ma reconnaissance va vers mon épouse Florence et mes enfants, Théophile, Suzanne et Nicolas, qui m'ont soutenu et supporté durant ces trois années, riches de rebondissements pour eux aussi. Florence, en particulier, a été très courageuse en effectuant des relectures multiples des publications et du présent manuscrit.

Table des matières

Liste des tables

Liste des figures

Liste des algorithmes

Introduction

Bob prend son téléphone mobile ce soir là. Il veut inviter Alice à dîner. Il compose le numéro de son smartphone et obtient la communication. Il est heureux de ce qu'Alice lui fait bon accueil et accepte avec joie son invitation pour le samedi suivant. Il lui reste cependant à prévoir le menu. Hélas, il n'aura pas le temps de faire les courses et il décide donc de commander en ligne auprès de sa grande surface préférée. Au menu, il y aura de la langouste accompagnée d'un riz pilaf et une petite salade gourmande. Alice ayant proposé d'amener le dessert, il termine sa commande sans oublier d'ajouter le Riesling et de cocher l'option livraison à domicile, et paie en ligne à l'aide de sa carte de crédit. Content de son début de soirée, Bob allume son téléviseur pour regarder la finale de la coupe de football sur la chaîne payante qui la diffuse.

Ce petit épisode de la vie de notre ami Bob est somme toute banal. La plupart de nos contemporains ne prêtent aucune attention particulière à ce genre d'événements. Et pourtant, comme monsieur Jourdain faisait de la prose sans le savoir, Bob s'est montré un utilisateur intensif d'outils de cryptographie, probablement sans en être conscient. Qu'on en juge plutôt : la communication téléphonique de Bob est cryptée (chiffrement A5/GSM) après un protocole d'authentification pour la connexion au réseau ; les courses et le paiement sont sécurisés également par des protocoles adaptés aux transactions commerciales en ligne, ceux-là considérés comme robustes, et on ne s'en plaindra pas ; enfin, la diffusion des programmes par chaîne payante fait aussi l'objet d'un cryptage. La cryptographie est l'ensemble des principes, techniques, moyens ou méthodes qui rendent inintelligibles des informations pour des tiers non autorisés. Elle s'invite régulièrement dans notre quotidien, et de façon naturelle de sorte que peu de gens y prennent garde et que beaucoup l'ignorent même. La sécurité et la confidentialité ou protection des données privées sont admises par tout un chacun de façon implicite.

Cette situation est récente, tandis que la cryptographie est une science ancienne. L'usage de chiffres est attesté en effet dès la plus haute Antiquité, en fait quasiment dès l'apparition de l'écriture : l'utilisation de mots de passe, de phrases codées en sont la forme balbutiante, et le célèbre code César tire son nom d'un de ses plus illustres utilisateurs, le grand Jules lui-même ! Ces chiffres vont se perfectionner et serviront lorsque la nécessité du secret est incontournable. Les correspondances militaires emploieront des chiffrements assez faibles dans le cas d'usage à court terme, pour les ordres tactiques lors des batailles par exemple ; les diplomates et les espions utiliseront, eux, des techniques plus sophistiquées. Avec le développement des moyens de communications à partir du XIX$^{\text{ème}}$ siècle, les besoins vont toutefois commencer à croître et les secteurs bancaire et commercial vont alors être également demandeurs. Tout au long de l'histoire et jusqu'aux années soixante-dix, la difficulté est alors liée à l'emploi de protocoles de chiffrement/déchiffrement symétriques, c'est à dire qui nécessitent l'échange d'une clé unique de chiffrement entre les deux parties. Cette contrainte coûteuse et risquée cantonnera l'usage de la cryptographie aux acteurs les plus importants de ces secteurs dont les états, les armées, les grandes banques ou sociétés multinationales. La plupart du temps, l'échange et la gestion des clés secrètes sont confiés à des sociétés spécialisées, pour le secteur civil en particulier.

En 1976, un tournant majeur se produit lorsque Diffie et Hellman dans [17] proposent un

protocole d'échange de clé de chiffrement qui peut se passer d'un canal de communication préalablement sécurisé. Cette avancée est suivie de près par une autre, en 1978, lorsque Rivest, Shamir et Adleman dans [54] élaborent le protocole de chiffrement RSA (ainsi nommé d'après leurs initiales). Avec ce protocole, le chiffrement/déchiffrement est dit asymétrique car il met en jeu deux clés différentes : une clé publique pour le chiffrement, connue de tous, et utilisable par chacun pour chiffrer un message que, seul, le destinataire sera en mesure de déchiffrer à l'aide d'une clé secrète, qu'il est le seul à connaître.

À partir de la fin des années soixante-dix, le développement des communications par Internet va s'appuyer sur ces nouvelles techniques qui changent radicalement la donne. Les besoins en communications chiffrées concernent maintenant tout un chacun et vont exploser en volume. Dans notre scène d'ouverture, Bob n'est qu'un parmi des centaines de millions d'usagers de l'Internet. De plus, la proportion de communications cryptées elle-même ne cesse d'augmenter. Lequel d'entre nous se passe aujourd'hui de tous les services en ligne, ne serait-ce que la messagerie électronique ? Ou encore, l'utilisateur des réseaux sociaux n'est-il pas demandeur de protection forte pour sa vie privée ?

Penchons-nous sur les protocoles de chiffrement asymétrique. L'utilisation de clés publiques présente l'avantage d'être souple tout en étant sûre. Pour séduisant qu'est ce procédé, il s'avère gourmand en calcul, et donc peu adapté à des communications intensives ou en temps réel, comme le chiffrement de communications téléphoniques, par exemple. De même, dans des applications embarquées, la limitation de la puissance des appareils risque de poser problème. C'est le cas de l'authentification des cartes à puces, par exemple, dont le développement en France dès les années 1980 a eu l'importance que l'on sait. Pour illustrer les difficultés techniques relatives à cette limitation, l'utilisation d'un protocole d'authentification simplifié au début de l'ère des cartes à puces s'est avéré désastreux, permettant à un ingénieur, Serge Humpich, de fabriquer des *Yes cards*, qui, pour fausses qu'elles étaient, s'avéraient dans certains cas reconnues comme d'authentiques cartes de crédit par les terminaux de paiement.

Plusieurs évolutions sont à noter dans les années 1980. En 1985, Peter Montgomery dans [46] propose une approche nouvelle pour les multiplications modulaires sur les grands entiers, ce qui accélère l'usage du protocole RSA qui commence à se populariser. La même année, Victor Miller dans [45] et Neal Koblitz dans [36] proposent indépendamment l'utilisation de protocoles basés sur les courbes elliptiques, avec pour objectif de réduire la taille des données pour un niveau de sécurité équivalent à RSA, et c'est ainsi que depuis lors, les protocoles basés sur les courbes elliptiques se répandent. Dans les années qui suivent, on assiste à la diffusion de bibliothèques dédiées aux protocoles cryptographiques, comme OpenSSL qui voit le jour en 1998. D'autres améliorations vont compléter les implantations des opérations arithmétiques employées dans ces protocoles, rendant l'emploi de la cryptographie quasiment invisible pour l'utilisateur final, tout en augmentant le niveau de sécurité.

Nous examinons trois catégories d'opérations arithmétiques auxquelles font appel la plupart de ces protocoles : l'exponentiation modulaire sur corps ou anneau finis, la multiplication scalaire de point de courbe elliptique sur corps fini de grande caractéristique, et la même opération sur un point de courbe elliptique sur corps binaire. Concernant l'exponentiation modulaire, l'état de l'art s'articule autour de l'emploi de la multiplication modulaire de Montgomery. Cette technique qui date de la fin des années 1980 a été perfectionnée grâce à des approches par mots machines comme celles présentées par Bosselaers *et al.* dans [11]. Ces méthodes classiques constituent un bon point de départ pour élaborer des perfectionnements.

En ce qui concerne la multiplication scalaire de point de courbe elliptique sur corps fini de grande caractéristique, l'implantation logicielle a été étudiée par différents auteurs. Sur l'arithmétique du corps, les travaux de Brown *et al.* dans [12] ainsi que la synthèse de Guajardo *et al.* dans [20] permettent d'implanter efficacement ces opérations. Dans le cas des extensions de

corps binaires, les travaux de Hankerson *et al.* dans [22] pour l'implantation logicielle, complétés en particulier par ceux de Lopez et Dahab sur la multiplication de polynômes binaires dans [43] ont constitué le socle de certaines de nos réflexions. La recherche a continué sur la caractéristique deux avec plus récemment l'approche parallèle de Taverne *et al.* dans [64, 65]. Concernant les opérations sur points de courbe elliptique, on trouve dans [3] une base de données complète sur ces opérations pour différents types de courbes, elliptiques et hyperelliptiques. Sur l'ensemble de la question de la multiplication scalaire de point de courbe elliptique indépendamment du corps fini, les approches classiques *Double-and-add* et toutes les variantes qui en découlent, ainsi que l'échelle binaire de Montgomery, sont présentées aussi par Hankerson *et al.* dans [22]. Tous ces travaux permettent de réaliser des implantations logicielles qui sont au niveau de l'état de l'art en matière de performances pour les approches considérées.

Parallèlement, la cryptanalyse, c'est-à-dire les attaques permettant à un adversaire de déchiffrer les communications cryptées, a également bénéficié d'avancées sensibles. Les algorithmes permettant de résoudre les problèmes difficiles qui sous-tendent la sécurité des protocoles cryptographiques font l'objet de recherches de la part de nombreux mathématiciens et informaticiens. À ce jour, pour les principaux problèmes en question, la complexité de leur résolution reste compatible avec les niveaux de sécurité exigés. Cependant, dans les années 1990, de nouvelles attaques ont été proposées, exploitant des sources d'informations supplémentaires par le biais de ce que l'on appelle les canaux auxiliaires (en anglais *side-channels*). Ces attaques, dont celles présentées par Kocher *et al.* dans [37, 38, 39], exploitent des informations collectées à l'aide d'appareils de mesures physiques de grandeurs caractérisant le fonctionnement des calculateurs cryptographiques, tels que le rayonnement électromagnétique ou le courant instantané consommé. Elle permettent ainsi de contourner les problèmes difficiles que nous avons évoqués plus haut. Ces attaques se perfectionnent au fil du temps et constituent aujourd'hui une vraie menace dès lors que l'adversaire accède physiquement à la machine à laquelle il s'intéresse. Kocher *et al.* dans [39] présentent en particulier une attaque déjà ancienne et l'une des plus connues, l'attaque *Simple Power Analysis*.

Dans ce travail de thèse, nous nous sommes consacrés à l'élaboration d'algorithmes pour le calcul d'opérations arithmétiques utilisées dans les principaux protocoles de cryptographie asymétrique et à leur implantation logicielle efficace, en vue d'améliorer leur performance tout en leur conférant une résistance à l'attaque *Simple Power Analysis*.

Organisation de la thèse

La thèse est organisée de la manière suivante : dans la première partie, nous faisons une revue de l'état de l'art en matière de calculs arithmétiques utilisés dans les protocoles cryptographiques ; dans la deuxième partie nous présentons nos contributions principales.

La première partie comprend trois chapitres. Au chapitre 1, nous donnons des rappels sur les principaux algorithmes de calculs des opérations élémentaires sur les anneaux et corps finis. Sur $\mathbb{Z}/N\mathbb{Z}$ et $\mathbb{F}_p$, nous exposons les opérations d'addition, de soustraction, de multiplication et d'élévation au carré multiprécisions, puis l'approche de Montgomery pour la réduction modulaire. Sur $\mathbb{F}_p$ dans le cas où p a une forme de nombre premier de Mersenne, pseudo-Mersenne ou Mersenne généralisé, nous présentons les méthodes efficaces de réduction modulaire. Nous présentons aussi l'opération d'inversion modulaire. Sur corps binaire $\mathbb{F}_{2^m}$, nous revenons brièvement à la construction de ces corps tout en donnant également des rappels sur les opérations élémentaires, addition et soustraction, les deux approches principales de multiplications de polynômes, la réduction modulo le polynôme irréductible constituant la base polynomiale du corps, le calcul de racine carrée et l'inversion d'un élément de ce corps.

Au chapitre 2, nous débutons par la présentation du protocole de chiffrement RSA basé sur

le problème de la factorisation des grands entiers, puis trois protocoles basés sur le logarithme discret : l'échange de clés de Diffie-Hellman, le chiffrement El Gamal et les protocoles de signature *Digital Signature Algorithm* et *Elliptic Curve Digital Signature Algorithm* (DSA et ECDSA). Nous présentons ensuite les deux algorithmes classiques d'exponentiation modulaire utilisés dans ces protocoles, les *Right-to-left Square-and-multiply* et *Left-to-right Square-and-multiply*. Puis, nous présentons les courbes elliptiques sur corps fini $\mathbb{F}_p$ et $\mathbb{F}_{2^m}$, notamment avec les additions et doublements de points dans différents systèmes de coordonnées projectives, ainsi que l'opération de *halving*, soit la division par deux d'un point d'ordre impair dans le cas d'une courbe sur corps fini de caractéristique deux. Nous abordons pour finir la multiplication scalaire de point de courbe elliptique avec les approches *Double-and-add* et ses variantes, notamment par l'utilisation de différents types de représentation du scalaire, les *Non Ajacent Form* et *Window Non Ajacent Form* (NAF et γ-NAF). Dans le cas de la caractéristique deux, nous présentons également l'approche *Halve-and-add* et celle proposée par Taverne *et al.* dans [64, 65], *Parallel Double/halve-and-add* .

Au chapitre 3, nous abordons la question des attaques par canal auxiliaire. Nous abordons le classement des principales attaques par canal auxiliaire sur la base des travaux de Zhou *et al.* dans [67] et de Verbauwhede *et al.* dans [66]. Nous détaillons ensuite trois attaques plus précisément avec quelques contre-mesures parmi les plus connues sur la base des travaux de Kocher *et al.* dans [37, 38, 39]. En premier lieu, nous présentons l'attaque *Simple Power Analysis* ainsi que quelques-unes des principales contre-mesures, à savoir l'algorithme *Double-and-add-always* et sa variante pour l'exponentiation modulaire, l'algorithme d'exponentiation modulaire *Square-always* de Clavier *et al.* dans [15], l'échelle binaire de Montgomery, l'algorithme régulier de multiplication scalaire *Double-add* de Joye dans [30] et les deux algorithmes réguliers *Regular left-to-right 2^γ-ary Exponentiation* et *Regular right-to-left 2^γ-ary Exponentiation* proposés par Joye et Tunstall dans [31]. Nous passons ensuite la *Timing Attack* et nous terminons par la présentation de l'attaque *Differential Power Analysis* ainsi que trois techniques de randomisation du scalaire présentées par Coron dans [16] qui en constituent les contre-mesures classiques.

La deuxième partie comprend quatre chapitres. Au chapitre 4, nous proposons des algorithmes optimisant les opérations combinées de type $A \cdot B, A \cdot C$ et $AB_1, \ldots, AB_\ell$, soient des multiplications modulaires de Montgomery partageant un opérande commune, dans un anneau fini $\mathbb{Z}/N\mathbb{Z}$. Nous utilisons ces opérations améliorées dans les algorithmes réguliers d'exponentiation modulaire résistants à l'attaque *Simple Power Analysis*, dans le but de les rendre plus efficaces : l'échelle binaire de Montgomery et les deux algorithmes *Regular left-to-right 2^γ-ary exponentiation* et *Regular right-to-left 2^γ-ary exponentiation* proposés par Joye et Tunstall dans [31]. Nous montrons ensuite les résultats de performances des implantations logicielles d'exponentiations modulaires utilisant ces approches, de type déchiffrement RSA pour des tailles de 1024, 2048 et 4096 bits, et qui montrent des gains de performance significatifs.

Au chapitre 5, nous étudions l'impact d'opérations combinées de type $A \cdot B, A \cdot C$ et $A \cdot B + C \cdot D$ sur corps binaire $\mathbb{F}_{2^m}$ aux additions et doublements de points de courbe elliptique sur corps binaires. Nous utilisons ensuite ces opérations de points dans les algorithmes de multiplication scalaire suivants : *Double-and-add*, *Halve-and-add* et *Double/halve-and-add* (approche parallèle de Taverne *et al.* dans [64, 65]) employant la réécriture du scalaire *γ-NAF*, ainsi que l'échelle binaire de Montgomery. Les implantations logicielles de mutiplications scalaires sur deux des courbes elliptiques sur corps binaire recommandées par le *National Institute of Standards and Technology* ([19], NIST), les B233 et B409, tirent parti de ces approches et montrent également des gains de performances.

Au chapitre 6, nous proposons un nouvel algorithme d'échelle binaire de Montgomery basé sur l'opération de *halving*, présentant les propriétés conférant une résistance face à l'attaque *Simple Power Analysis*, ainsi qu'à la *safe-error attack*. Nous présentons ensuite une approche

parallèle de l'échelle binaire de Montgomery appliquée à la multiplication de point de courbe elliptique sur corps binaire dans l'esprit de celle de Taverne *et al.* dans [64, 65] utilisant l'échelle binaire de Montgomery classique basée sur le doublement de point ainsi que notre approche à base de *halving*. Nous finissons par les performances de ces approches et leur amélioration en comparaison de l'échelle binaire de Montgomery classique, sur les deux courbes elliptiques B233 et B409, et deux plates-formes différentes, dont une mobile.

Au chapitre 7, notre étude porte sur l'implantation logicielle de l'approche parallèle de l'algorithme *Double-and-add* présentée par Moreno et Hasan dans [48] dans le but de conférer de la résistance face à l'attaque *Simple Power Analysis*. Nous proposons un mécanisme de synchronisation des *threads* permettant de rendre efficace cette approche. Nous avons implanté les versions suivantes : *Parallel NAF Double-and-add* à deux *threads* sur courbe sur corps binaires B233 et B409, sur une courbe elliptique de Weierstrass et une courbe Jacobi Quartic sur corps $\mathbb{F}_p$ avec $p = 2^{255} - 19$, *Parallel NAF Halve-and-add* à deux *threads* et *Parallel NAF Double-halve-and-add* à quatre *threads* pour les courbes B233 et B409. Nous montrons les résultats de performance et leur amélioration par rapport aux approches séquentielles et parallèles de l'état de l'art correspondantes.

Résumé des contributions

Les principales contributions de cette thèse sont les suivantes :

— deux algorithmes de multiplications modulaires de Montgomery multiples partageant un opérande commune, et l'application des opérations précédentes à trois approches d'exponentiation modulaire régulière résistante à l'attaque SPA ([50], travail publié dans les actes de la conférence Arith 22, Lyon, juin 2015) ;

— deux algorithmes d'opérations combinées pour AB, AC et $AB + CD$ sur corps binaires, adaptés chacun à une plate-forme spécifique, ainsi que l'étude de l'impact des opérations précédentes sur les approches de multiplication scalaire et les gains de performances qui en résultent ([51], travail publié dans les actes de la conférence AfricaCrypt, Le Caire, juin 2013) ;

— l'élaboration d'une nouvelle échelle binaire de Montgomery basée sur l'opération de *halving*, et l'étude de l'impact sur les performances de son emploi dans une version parallèle ([52], travail publié dans la revue *IEEE-Transactions on Computer*, 2015) ;

— l'implantation logicielle efficace d'une approche parallèle basée sur l'algorithme *Right-to-left Double-and-add*, son extension aux approches *Halve-and-add* et *Double/halve-and-add* ([55], travail publié dans les actes de la conférence InsCrypt, Beijing, décembre 2014).

Première partie

État de l'art

Chapitre 1

Arithmétique des anneaux $\mathbb{Z}/N\mathbb{Z}$ et corps finis $\mathbb{F}_p$ et $\mathbb{F}_{2^m}$

Dans le cas des applications cryptographiques, nous avons besoin de construire des anneaux ou des corps premiers. C'est le cas de plusieurs protocoles (RSA, échange de clé...).

Pour assurer la sécurité, les calculs vont se faire sur des grands nombres. Prenons l'exemple du protocole de chiffrement RSA (du nom de leurs auteurs Rivest, Shamir et Adleman dans [54]). Les normes indiquent des tailles de 2048 bits pour les nombres que l'on manipule, soit un ordre de grandeur de 10^{617}. C'est l'ordre de grandeur de $N = pq$ avec p et q premiers, le module qui définit l'anneau. Ceci entraîne par conséquent une taille moitiée pour p et q. Il est donc permis de parler de grands nombres ! Certains auteurs font d'ailleurs remarquer que même des domaines comme l'astrophysique ou la physique des particules ne manipulent pas d'aussi grands nombres. Pour illustrer ce fait, on peut simplement mentionner l'évaluation cosmologique du nombre total d'atomes dans l'univers visible, qui est de 10^{80}, seulement...

Pour effectuer ces calculs, nous aurons donc besoin d'un ordinateur et de quelques bons algorithmes de calculs sur de tels nombres. Dans ce chapitre, après de brefs rappels sur l'anneau $(\mathbb{Z}/N\mathbb{Z}, +, 0, \times, 1)$, nous donnons les algorithmes d'addition/soustraction, puis nous nous intéressons à la multiplication modulaire et aux différentes variantes de réduction modulaire. Nous terminons enfin par l'inversion modulaire, avec les particularités relatives au corps $\mathbb{F}_p$. Nous suivons ensuite le même plan pour les corps binaires $\mathbb{F}_{2^m}$.

1.1 Opérations sur l'anneau des entiers modulo N

1.1.1 Quelques rappels sur l'anneau $(\mathbb{Z}/N\mathbb{Z}, +, 0, \times, 1)$

L'ensemble $\mathbb{Z}$ est constitué par les nombres entiers positifs, négatifs et nul. C'est un ensemble infini et c'est un anneau commutatif unitaire, lorsqu'on le munit de l'addition (0 est élément neutre) et de la multiplication (1 est élément neutre).

On remarque qu'un élément $A \in \mathbb{Z}$ peut s'écrire $A = QN + R$ avec $0 \leq R \leq N - 1$. Les entiers Q et R sont respectivement le quotient et le reste de la division euclidienne de A par N. On dit que deux entiers A et B (éléments de $\mathbb{Z}$) sont congrus modulo N si les restes de leur division euclidienne par N sont égaux entre eux. Par exemple, 8 et 13 sont congrus modulo 5, les restes de leur division sont identiques, en l'occurrence égaux à 3.

On peut construire l'anneau quotient $(\mathbb{Z}/N\mathbb{Z}, +, 0, \times, 1)$ en utilisant la relation de congruence modulo N entre les entiers (éléments de $\mathbb{Z}$), c'est-à-dire s'ils sont congrus modulo N. C'est une relation d'équivalence. $(\mathbb{Z}/N\mathbb{Z}, +, 0, \times, 1)$, qu'on note $\mathbb{Z}/N\mathbb{Z}$, comporte donc les N classes d'équivalences suivantes :

$$\{\bar{0}, \bar{1}, \bar{2}, \ldots, \overline{N-1}\} \text{ où } \bar{A} = \{A + \alpha N, \alpha \in \mathbb{Z}\}.$$

Pour tout entier $A \in \mathbb{Z}$, il existe $R \in \{0, 1, \ldots, N-1\}$ tel que $A \equiv R \pmod{N}$. Dans la suite, une classe d'équivalence $\bar{A}$ sera représentée par l'entier R. C'est la forme réduite de A.

Si N est premier, alors $\mathbb{Z}/N\mathbb{Z}$ est un corps, c'est à dire que tous ses éléments non nuls sont inversibles pour la multiplication. Si on pose $p = N$, alors on le note $\mathbb{F}_p$.

1.1.2 Addition/soustraction modulaire

Soient A, B deux éléments de $\mathbb{Z}/N\mathbb{Z}$, représentés sous forme réduite. On a donc $0 \leq A, B < N$. On veut calculer $S = A + B \bmod N$.

Lorsqu'on effectue une addition modulaire, on souhaite obtenir le résultat sous forme réduite également. La façon courante de procéder est de comparer le résultat à N et de soustraire N le cas échéant.

Si on suppose que les opérandes A et B sont stockées sur n mots machine, on doit effectuer les additions avec retenue. Ceci conduit à l'algorithme 1.1.

Algorithme 1.1 Addition modulaire dans $\mathbb{Z}/N\mathbb{Z}$

Require: $A, B < N$ de taille n mots machine, respectivement $\{a_{n-1}, a_{n-2}, \ldots, a_1, a_0\}$, $\{b_{n-1}, b_{n-2}, \ldots, b_1, b_0\}$ et $\{N_{n-1}, N_{n-2}, \ldots, N_1, N_0\}$.
Ensure: $S = A + B \bmod N$
1: $c \leftarrow 0$
2: **for** $i = 0$ to $n - 1$ **do**
3: $\quad s_i \leftarrow \text{ADDC}(a_i, b_i, c), c \leftarrow carry$ // addition avec retenue
4: **end for**
5: **if** $S \geq N$ **then**
6: $\quad c \leftarrow 0$
7: $\quad$ **for** $i = 0$ to $n - 1$ **do**
8: $\quad\quad s_i \leftarrow \text{SBB}(s_i, N_i, c), c \leftarrow borrow$ // soustraction avec retenue
9: $\quad$ **end for**
10: **end if**
11: **return** $S = \{s_{n-1}, s_{n-2}, \ldots, s_1, s_0\}$

De manière symétrique, on construit la soustraction modulaire, présentée dans l'algorithme 1.2.

1.1.3 Multiplication modulaire dans $\mathbb{Z}/N\mathbb{Z}$

1.1.3.1 Multiplication et élévation au carré multiprécision

Nous abordons ici la multiplication multiprécision. On se souvient de nos leçons d'école primaire. Nous utiliserons cette bonne vieille méthode que les anglophones baptisent *schoolbook*. La différence concerne ce que l'on appelle un mot. Dans notre enfance, notre mot machine, pour nous autres humains, est compris entre 0 et 9, le chiffre en base 10. Les ordinateurs, eux, manipulent des mots machine dont la taille est de 8, 16 bits pour les machines anciennes, 32 ou 64 bits sur les machines les plus modernes.

Nous présentons cet algorithme dans le cas d'une multiplication avec des opérandes de tailles différentes : l'algorithme 1.3 multiplie un nombre de taille 1 mot machine avec une opérande de taille n mots machine, et où w est la taille d'un mot en bits.

La table 1.1 donne l'évaluation étape par étape de la complexité de l'algorithme 1.3.

Algorithme 1.2 Soustraction modulaire dans $\mathbb{Z}/N\mathbb{Z}$

Require: $A, B < N$ de taille n mots machine, respectivement $\{a_{n-1}, a_{n-2}, \ldots, a_1, a_0\}$, $\{b_{n-1}, b_{n-2}, \ldots, b_1, b_0\}$ et $\{N_{n-1}, N_{n-2}, \ldots, N_1, N_0\}$.
Ensure: $S = A - B \bmod N$
1: $c \leftarrow 0$
2: **for** $i = 0$ to $n - 1$ **do**
3: $s_i \leftarrow \text{SBB}(a_i, b_i, c), c \leftarrow borrow$ // soustraction avec retenue
4: **end for**
5: **if** $S < 0$ **then**
6: $c \leftarrow 0$
7: **for** $i = 0$ to $n - 1$ **do**
8: $s_i \leftarrow \text{ADDC}(s_i, N_i, c), c \leftarrow carry$ // addition avec retenue
9: **end for**
10: **end if**
11: **return** $S = \{s_{n-1}, s_{n-2}, \ldots, s_1, s_0\}$

Algorithme 1.3 Multiplication multiprécision (opérandes $1 \times n$ mots)

Require: A, b avec $A = (a_{n-1}, \ldots, a_0)_{2^w}$ et $0 \leq b, a_i < 2^w$.
Ensure: $X = (x_n, \ldots, x_0)_{2^w}$ avec $X = A \cdot b < 2^{(n+1)w}$
1: $(v, u) \leftarrow b \cdot a_0$ //u est la partie basse et v la partie haute
2: $x_0 \leftarrow u$, $x_1 \leftarrow v$
3: $c \leftarrow 0$
4: **for** $j = 1$ to $n - 1$ **do**
5: $(v, u) \leftarrow b \cdot a_j$ //u est la partie basse et v la partie haute
6: $x_i \leftarrow \text{ADDC}(x_i, u, c), c \leftarrow carry$ // addition avec retenue
7: $x_{i+1} \leftarrow v$
8: **end for**
9: $x_n \leftarrow \text{ADDC}(x_n, 0, c)$ // absorption de la dernière retenue
10: **return** $X = (x_n, \ldots, x_0)_{2^w}$

	Opérations	# ADD	# MUL
étape 1	$b \cdot a_0$	-	1
$n-1$ étapes 5	$b \cdot a_j$	-	$n-1$
$n-1$ étapes 6	ADDC(x_i, u)	$n-1$	-
étape 9	ADDC$(x_n, 0)$	1	-
Total		n	n

TABLE 1.1 – Complexité de la multiplication multiprécision (opérandes $1 \times n$ mots de w bits)

Quand les deux opérandes sont de la même taille de n mots machine, on utilise alors n fois l'algorithme précédent comme suit : soient A, B, avec $A = (a_{n-1}, \ldots, a_0)$, et $B = (b_{n-1}, \ldots, b_0), 0 \leq a_i, b_i < 2^w$ où w est la taille d'un mot, alors

$$A \cdot B = \sum_{i=0}^{n-1} A \times b_i \times 2^{wi}. \tag{1.1}$$

L'équation (1.1) conduit à l'algorithme 1.4, la table 1.2 donne l'évaluation étape par étape de sa complexité en fonction de celle de l'algorithme 1.3.

Algorithme 1.4 Multiplication multiprécision (opérandes de N mots)

Require: A, B, avec $A = (a_{n-1}, \ldots, a_0)$ et $B = (b_{n-1}, \ldots, b_0), 0 \leq a_i, b_i < 2^w$ où w est la taille d'un mot.
Ensure: $X = A \cdot B = (x_{2n-1}, \ldots, x_0)$ avec $X < 2^{2nw}$
1: $x_i \leftarrow 0$ pour tout $i \in \{0, \ldots, 2n-1\}$
2: **for** $i = 0$ to $n-1$ **do**
3: $T \leftarrow A \times b_i$ // algorithme 1.3, la taille de T est $n+1$ mots
4: $c \leftarrow 0$
5: **for** $j = 0$ to n **do**
6: $x_{i+j} \leftarrow \text{ADDC}(x_{i+j}, t_j, c), c \leftarrow carry$
7: **end for**
8: **end for**
9: **return** (X)

	Opérations	# ADD	# MUL
n étapes 3	$A \cdot b_i$	n^2	n^2
$n(n+1)$ étapes 6	$\text{ADDC}(x_{i+j}, t_j, c)$	$n(n+1)$	-
Total		$2n^2 + n$	n^2

TABLE 1.2 – Complexité de la multiplication multiprécision (opérandes $n \times n$ mots de w bits)

Pour calculer un carré, on peut utiliser cet algorithme. Cependant, dans ce cas, on remarque que certaines multiplications élémentaires sont effectuées deux fois. En effet, si w est la taille du mot, pour $A = \sum_{i=0}^{n-1} a^i 2^{wi}$, on a :

$$A^2 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_i \cdot a_j \cdot 2^{w(i+j)}.$$

Dans cette formule, les multiplications $a_i \cdot a_j$ réapparaissent sous la forme $a_j \cdot a_i$. On peut récrire

cette formule comme suit :

$$\begin{aligned} A^2 &= \sum_{i=0}^{n-1} a_i^2 \cdot 2^{2wi} + 2 \cdot \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} a_i \cdot a_j \cdot 2^{w(i+j)} \\ &= \sum_{i=0}^{n-1} a_i 2^{wi} \times \left(a_i + 2 \sum_{j=i+1}^{n-1} a_j \cdot 2^{wj} \right) \\ &= \sum_{i=0}^{n-1} a_i 2^{wi} \times \widetilde{A}_i \end{aligned}$$

en posant $\widetilde{A}_i = a_i + 2\sum_{j=i+1}^{n-1} a_j \cdot 2^{wj}$. Cette écriture présente l'avantage de réduire le nombre total de multiplications et d'additions élémentaires. Ceci est présenté dans l'algorithme 1.5.

Algorithme 1.5 Élévation au carré multiprécision (opérande de n mots)

Require: A, avec $A = (a_{n-1}, \ldots, a_0), 0 \leq a_i < 2^w$ où w est la taille d'un mot.
Ensure: $X = A^2 = (x_{2n-1}, \ldots, x_0)$ avec $X < 2^{2nw}$
1: $A' \leftarrow A + A$
2: $\widetilde{A}_0 \leftarrow (a'_{n-1}, \ldots, a'_2, |2a_1|_{2^w}, a_0)_{2^w}$
3: $X \leftarrow \widetilde{A}_0 \cdot a_0$
4: **for** $i = 1$ to $(n-1)$ **do**
5: $\quad \widetilde{A}_i \leftarrow (a'_{n-1}, \ldots, a'_{i+2}, |2a_{i+1}|_{2^w}, a_i)_{2^w}$
6: $\quad X \leftarrow X + \widetilde{A}_i \cdot a_i \cdot 2^{wi}$
7: **end for**
8: **return** (X)

Les termes $\widetilde{A}_i$ sont de taille $n - i$ mots machine. Ceci permet d'évaluer la complexité de l'algorithme 1.5 :

- étape 1, c'est une addition multiprécision sur n mots, soient n additions de mots ;
- étape 2, on effectue une addition de mots sans retenue ($\mathrm{mod} 2^w$) ;
- étape 3, on a une multiplication $1 \times n$ mots, soient n additions et n multiplications de mots ;
- étape 5, c'est identique à l'étape 2, soit une addition de mots ;
- étape 6 :
  - on a $n-1$ multiplications avec un opérande de taille $n-i$ multipliée par un opérande d'un mot, soit $n - i$ additions et $n - i$ multiplications, ce qui fait au total au total $n(n-1)/2$ additions et multiplications de mots ;
  - étape 6 encore, on accumule ensuite le résultat de taille $n - i + 1$ mots de ces multiplications à chaque tour de boucle, soient $n - i + 1$ additions de mots lors de chacun des $n - 1$ tours de boucle, ce qui ajoute $n(n-1)/2 + n - 1$ additions de mots.
- pour l'étape 6, le total est donc $(n+1)(n-1)$ additions de mots et $n(n-1)/2$ multiplications de mots.

La table 1.3 synthétise cette évaluation étape par étape de la complexité de l'algorithme 1.5. La table 1.4 regroupe les complexités des opérations multiprécision que nous venons d'examiner.

Nous savons comment multiplier et élever au carré des nombres entiers, nous allons maintenant examiner comment intégrer la réduction modulaire dans la section suivante.

	Opérations	# ADD	# MUL
étape 1	$A + A$	n	-
étape 2	$\|2a_1\|_{2^w}$	1	-
étape 3	$\widetilde{A}_0 \cdot a_0$	n	n
$n-1$ étapes 5	$\|2a_{i+1}\|_{2^w}$	$n-1$	-
$n-1$ étapes 6	$X + \widetilde{A}_i \cdot a_i \cdot 2^{wi}$	$(n-1)(n+1)$	$\frac{n(n-1)}{2}$
Total		$n^2 + 3n - 1$	$\frac{n^2}{2} + \frac{n}{2}$

TABLE 1.3 – Complexité de l'élévation au carré (opérande de n mots de w bits)

Algorithme	# ADD	# MUL	Total
Multiplication $1 \times n$, algorithme 1.3	n	n	$2n$
Multiplication, algorithme 1.4	$2n^2 + n$	n^2	$3n^2 + n$
Carré, algorithme 1.5	$n^2 + 3n - 1$	$\frac{n^2}{2} - \frac{n}{2}$	$\frac{3}{2}n^2 + \frac{5n}{2} - 1$

TABLE 1.4 – Complexité des opérations multiprécision (opérandes de n mots de w bits)

1.1.3.2 Multiplication modulaire de Montgomery

La réduction modulo N d'un entier X consiste à calculer la forme réduite que nous avons déjà évoquée, c'est à dire le reste de la division de X par le module N. On écrit également

$$R = X - q \times N \text{ tel que } R < N$$

où R est le reste et q est le quotient. On remarque immédiatement que ce calcul est coûteux, celui d'une division multiprécision (pour obtenir q) suivi de la multiplication et de la soustraction correspondantes pour le calcul du reste.

Montgomery dans [46] suggère une autre façon d'effectuer la réduction modulaire. Nous la présentons dans les lignes qui suivent.

Soient $A, B, N < M$ des entiers, M et N étant premiers entre eux, et M tel que $M = 2^r$. Dans une réduction modulaire classique, en calculant le reste de la division, on «efface» les bits de poids forts et on ramène ainsi le résultat des multiplications à la taille voulue. L'idée de Montgomery, c'est d'effacer les r bits de poids faibles. Une fois ceci réalisé, une division par $M = 2^r$ nous permet de ramener le nombre à la taille voulue. On calcule :

$$q \leftarrow A \times B \cdot (-N^{-1}) \bmod M.$$

Alors, la division par M dans la formule suivante

$$X \leftarrow (A \times B + q \times N)/M.$$

est exacte, et on remarque que

$$X \equiv ABM^{-1} \bmod N \text{ avec } X < N. \tag{1.2}$$

Algorithme 1.6 Multiplication modulaire de Montgomery $MM(A, B)$

Require: $A, B, N < M$, $(-N^{-1}) \bmod M$ précalculé.
Ensure: $X = A \cdot B \cdot M^{-1} \bmod N$ avec $X < N$
1: $X \leftarrow A \cdot B \quad //(X < M \cdot N)$
2: $q \leftarrow X \times (-N^{-1}) \bmod M$
3: $X \leftarrow (X + q \times N)/M \quad //(X < 2 \cdot N)$
4: **if** $X > N$ **then**
5: $\quad X \leftarrow X - N \quad //(X < N)$
6: **end if**
7: **return** $(X \equiv ABM^{-1} \bmod N)$

L'algorithme 1.6 présente la multiplication de Montgomery $MM(A, B)$.

Cet algorithme est efficace dans la mesure où une division par $M = 2^r$ est un simple décalage à droite de r bits. Quant à la réduction modulo M, c'est un simple masquage des bits de poids forts pour ne conserver que les r bits de poids faibles.

Cependant, cette multiplication modulaire n'est intéressante que lorsqu'on effectue une séquence de plusieurs multiplications modulaires, en raison de la présence du facteur M^{-1}. Pour tenir compte de ce facteur, on recourt à l'utilisation de la représentation de Montgomery. Dans cette représentation, pour un élément A donné, on pose $\tilde{A} \leftarrow A \cdot M \bmod N$.

$$\begin{aligned} MM(\tilde{A}, \tilde{B}) &= A \cdot M \times B \cdot M \times M^{-1} \bmod N \\ &= AB \cdot M \bmod N \\ &= \widetilde{AB}. \end{aligned}$$

La conversion vers cette représentation s'effectue comme suit :

$$\begin{aligned} MM(A, M^2 \bmod N) &= A \cdot M^2 \times M^{-1} \bmod N \\ &= A \cdot M \bmod N \\ &= \tilde{A} \end{aligned}$$

et la conversion inverse comme suit :

$$\begin{aligned} MM(\tilde{A}, 1) &= \tilde{A} \times M^{-1} \bmod N \\ &= A \bmod N. \end{aligned}$$

1.1.3.3 Réduction, multiplication et élévation au carré de Montgomery par blocs

Nous présentons dans cette section les variantes de l'algorithme 1.6 travaillant mot à mot. L'intérêt de ces méthodes est de limiter la taille des variables manipulées à celle des opérandes d'entrée plus un. Le principe repose sur l'utilisation d'une réduction modulaire d'un mot, intercalée entre les étapes des algorithmes de multiplication ou d'élévation au carré, les algorithmes 1.4 et 1.5.

1.1.3.3.1 Petite réduction d'un mot (*smallRed*)

Le cœur de cet algorithme est une petite réduction d'un mot à la fois (*smallRed*). En entrée, nous avons $A < 2^w \cdot M$ et en sortie nous calculons $A = A2^{-w} \bmod N$ (voir l'algorithme 1.7).

La complexité de cet algorithme avec une entrée de taille n' mots est $\max(n', n + 1) + n$ ADD et $n + 1$ MUL. Le lemme 1.1.1 fournit deux résultats utiles dans la suite.

Lemme 1.1.1. *Soit $A \geq 0$ l'entrée de l'algorithme 1.7. Alors, le résultat X satisfait*

Algorithme 1.7 Petite réduction de Montgomery d'un mot à la fois (*smallRed*(A))

Require: Le module $N < 2^{wn-1}$ et un entier positif $A = (a_{n'-1}, \ldots, a_0)_{2^w}$ de taille n' mots machine, et $N' = (-N^{-1}) \bmod 2^w$.
Ensure: $X = A \cdot 2^{-w} \bmod N$ avec $X < A/2^w + N$
1: $q \leftarrow a_0 \cdot N' \bmod 2^w$
2: $X \leftarrow (A + q \cdot N)/2^w$
3: **return** X

i) $X < A/2^w + N$ *et* $X = A2^{-w} \bmod N$.

ii) Si $A < 2N$ *alors* $X < 2N$.

Démonstration. i) Par construction, on a $(A+q\cdot N) = 0 \bmod 2^w$. Ceci implique que la division par 2^w (étape 2 de l'algorithme 1.7) est exacte et X est donc entier. On a donc $X2^w = (A + q\cdot N) \equiv A \bmod N$ et aussi $X \equiv A\cdot 2^{-w} \bmod N$. De plus, on a $Y = (X+qN)/2^w < X/2^w+N$ puisque $q < 2^w$.

ii) Dans le cas où $A < 2N$, on a alors $X = (A + qN)/2^w < 2N/2^w + N < 2N$.

□

1.1.3.3.2 Multiplication modulaire de Montgomery mot à mot (*MontMul*)

Bosselaers *et al.* dans [11] donnent un algorithme complet de réduction de Montgomery. Dans le but d'éviter de grandes opérations multiprécision, on peut intégrer cette réduction mot à mot *smallRed* à la multiplication multiprécision (voir l'algorithme 1.4) et en déduire un algorithme de multiplication modulaire complet.

La façon de procéder consiste, dans l'algorithme 1.4 de multiplication $n \times n$, à insérer une réduction d'un mot (*smallRed*) entre chaque multiplication $1 \times n$, soit à la fin de chaque itération de la boucle **for** de l'algorithme, afin de maintenir l'accumulateur du résultat dans une taille de n mots. Nous présentons maintenant cette méthode.

Soient $A, B, N < M$ trois entiers, et M tel que $M = 2^{w\times n}$ où w est la taille (le nombre de bits) d'un mot machine. Comme dans l'algorithme 1.6, la multiplication de Montgomery mot à mot produit $X = A \cdot B \cdot M^{-1} \bmod N$ avec $X < N$. C'est l'algorithme 1.8.

Algorithme 1.8 *MontMul*, multiplication de Montgomery mot à mot [44]

Require: le module $N < 2^{wn-1}$, w la taille du mot machine en bits, $A = (a_{n-1}, \ldots, a_0)_{2^w}$ et $B = (b_{n-1}, \ldots, b_0)_{2^w}$ deux entiers sur n-mots dans $[0, N]$ et $N' = (-N^{-1}) \bmod 2^w$.
Ensure: $X = A \cdot B \cdot 2^{-wn} \bmod N$
1: $X \leftarrow a_0 \cdot B$
2: $q \leftarrow |X|_{2^w} \cdot N' \bmod 2^w$
3: $X \leftarrow (X + q \cdot N)/2^w$
4: **for** $i = 1$ to $n - 1$ **do**
5: $\quad X \leftarrow X + a_i \cdot B$
6: $\quad q \leftarrow |X|_{2^w} \cdot N' \bmod 2^w$
7: $\quad X \leftarrow (X + q \cdot N)/2^w$
8: **end for**
9: **if** $X > N$ **then**
10: $\quad X \leftarrow X - N$
11: **end if**
12: **return** X

Évaluons la complexité de l'algorithme 1.8. L'étape 3 est une simple multiplication puis addition d'opérandes de $n+1$ mots. Étapes 5 et 7, nous calculons deux multiplications $n \times 1$ combinées avec $n+1$ additions d'opérandes d'un mot. Par conséquent, le coût de ces deux étapes est de $2n$ multiplications d'opérandes d'un mot (on ne calcule qu'une seule fois $a_i b_0$) et $4n$ additions d'opérandes d'un mot.

Ceci est synthétisé par la table 1.5. Le coût de l'algorithme 1.8 est de $2n^2+n$ multiplications et $4n^2+2n-1$ additions d'opérandes d'un mot.

	Opérations	# ADD	# MUL
étape 1	$a_0 \times B$	n	n
étape 2	$\|X\|_{2^w} \cdot N'$	0	1
étape 3	$q \times N$	n	n
	$X + (qN)$	$n+1$	0
$n-1$ étapes 5	$a_i \times B$	$(n-1)n$	$(n-1)n$
	$X + (a_i B)$	$(n-1)(n+1)$	0
$n-1$ étapes 6	$\|X\|_{2^w} \cdot N'$	0	$n-1$
$n-1$ étapes 7	$q \times N$	$(n-1)n$	$(n-1)n$
	$X + (qN)$	$(n-1)(n+1)$	0
étape 10	$X - N$	n	0
Total		$4n^2+2n-1$	$n(2n+1)$

TABLE 1.5 – Complexité de l'algorithme 1.8 (*MontMul*)

1.1.3.3.3 Élevation au carré de Montgomery mot à mot (*MontSq*)

Il est possible d'étendre ce concept à l'élévation au carré, en gardant l'avantage de la complexité inférieure en nombre de multiplications par rapport à la multiplication multiprécision. Nous présentons ceci dans l'algorithme 1.9. Comme précédemment, la démarche consiste à insérer dans l'algorithme 1.5 des petites réductions *smallReds* pour conserver les résultats intermédiaires sur n mots. C'est ce que l'on voit dans les étapes 4 et 5, c'est à dire le prologue de l'algorithme et ensuite étapes 9 et 10, soit la boucle principale elle-même.

Sa complexité globale correspond à celle de l'algorithme 1.5 plus n complexités de *smallRed*, soit $3n^2+5n-1$ additions de mots et $\frac{3n^2}{2}+\frac{3n}{2}$ multiplications de mots. La table 1.6 donne l'évaluation étape par étape de la complexité de l'algorithme 1.9.

1.1.3.3.4 Complexités des opérations de Montgomery mot à mot

On trouvera dans la table 1.7 les complexités de toutes les opérations de Montgomery. On remarque que la complexité de *MontSq* est meilleure que celle de *MontMul* d'environ 25 %. Ceci est à attribuer à l'amélioration de l'élévation au carré multiprécision par rapport à la multiplication.

Algorithme 1.9 *MontSq*, élévation au carré de Montgomery mot à mot

Require: A, avec $A = (a_{n-1}, \ldots, a_0)_{2^w}$ et $0 \leq a_i < 2^w$ où w est la taille d'un mot machine, $N' = -N^{-1} \bmod 2^w$.
Ensure: $X \equiv A^2 \times 2^{-wn} \bmod N$ et $X < N$
1: $A' \leftarrow A + A$
2: $\widetilde{A}_0 \leftarrow (a'_{n-1}, \ldots, a'_2, |2a_1|_{2^w}, a_0)_{2^w}$
3: $X \leftarrow \widetilde{A}_0 \cdot a_0$
4: $q \leftarrow |X|_{2^w} \cdot N' \bmod 2^w$
5: $X \leftarrow (X + q \cdot N)/2^w$
6: **for** $i = 1$ to $(n-1)$ **do**
7: $\quad \widetilde{A}_i \leftarrow (a'_{n-1}, \ldots, a'_{i+2}, |2a_{i+1}|_{2^w}, a_i)_{2^w}$
8: $\quad X \leftarrow X + \widetilde{A}_i \cdot a_i \cdot 2^{wi}$
9: $\quad q \leftarrow |X|_{2^w} \cdot N' \bmod 2^w$
10: $\quad X \leftarrow (X + q \cdot N)/2^w$
11: **end for**
12: **if** $X > N$ **then**
13: $\quad X \leftarrow X - N$
14: **end if**
15: **return** X

	Opérations	# ADD	# MUL
étape 1	$A + A$	n	-
étape 2	$\|2a_1\|_{2^w}$	1	-
étape 3	$\widetilde{A}_0 \cdot a_0$	n	n
étape 4	$\|X\|_{2^w} \cdot N'$	-	1
étape 5	$q \times N$	n	n
	$X + (qN)$	$n+1$	-
$n-1$ étapes 7	$\|2a_{i+1}\|_{2^w}$	$n-1$	-
$n-1$ étapes 8	$X + \widetilde{A}_i \cdot a_i \cdot 2^{wi}$	$(n-1)(n+1)$	$\frac{n(n-1)}{2}$
$n-1$ étapes 9	$\|X\|_{2^w} \cdot N'$	-	$n-1$
$n-1$ étapes 10	$q \times N$	$(n-1)n$	$(n-1)n$
	$X + (qN)$	$(n-1)(n+1)$	-
étape 13	$X - N$	n	0
Total		$3n^2 + 5n - 1$	$\frac{3n^2}{2} + \frac{3n}{2}$

TABLE 1.6 – Complexité de l'algorithme 1.9 (*MontSq*)

Opération	# ADD	# MUL
smallRed (Algo. 1.7) entrée de taille n'-mots	$\max(n', n+1)+n$	$n+1$
MontMul (Algo. 1.8)	$4n^2+2n-1$	$2n^2+n$
MontSq (Algo. 1.9)	$3n^2+5n-1$	$\frac{3n^2}{2}+\frac{3n}{2}$

TABLE 1.7 – Complexité de *smallRed*, multiplication et carré de Montgomery.

1.1.3.4 Réduction modulaire dans le cas particulier des modules premiers de Mersennes et variantes

La multiplication modulaire de Montgomery que nous venons de voir avait pour objet d'accélérer les calculs pour un module quelconque, sans propriétés particulières. Cependant, dans le cas où nous avons un module qui peut être choisi librement, la réduction modulaire peut tirer avantage d'une forme particulière de ce nombre, notamment si ce nombre a une représentation binaire creuse (comportant une grande proportion de zéros). Dans certaines applications, on choisit alors pour ce module un nombre premier présentant cette caractéristique. C'est le cas des nombres de Mersenne, pseudo-Mersenne ou Mersenne généralisés. On rappelle ici ce que sont ces nombres :

— un nombre premier de Mersenne est de la forme 2^k-1 ;
— un nombre premier pseudo-Mersenne est de la forme $2^k-c, c\in\mathbb{N}, c$ petit ;
— un nombre premier de Mersenne généralisé est de la forme $\sum_{n_i} 2^{n_i k}$, k la taille d'un mot, $n_{max}k$ la taille du nombre premier.

1.1.3.4.1 Nombres premiers de Mersenne

La réduction modulaire d'un entier A par un nombre de Mersenne $p=2^k-1$ peut être calculée très rapidement de la façon suivante : on écrit

$$A = A_H\cdot 2^k + A_L,$$

puis, on tire parti de ce que $2^k\equiv 1 \bmod p$ par définition de p en récrivant A comme suit :

$$A \bmod p \equiv A_H + A_L \bmod p.$$

Ainsi, dans le cas d'un nombre premier de Mersenne, la réduction modulaire se résume à une addition modulaire. Le seul inconvénient, c'est que les nombres de Mersenne premiers sont peu nombreux. Le NIST, dans ses préconisations, en retient un : $P521=2^{521}-1$.

1.1.3.4.2 Nombres premiers pseudo-Mersenne

Dans le cas d'un pseudo-Mersenne, la liberté donnée par c permet de trouver de nombreux nombres premiers candidats. On a maintenant pour $p=2^k-c$ la relation $2^k\equiv c \bmod p$. La réduction modulaire se calcule alors :

$$A \bmod p \equiv c\cdot A_H + A_L \bmod p.$$

Dans la littérature, certains auteurs utilisent de tels nombres, comme par exemple Bernstein dans [8]. Il présente une implantation logicielle de calculs sur courbe elliptique sur $\mathbb{F}_p$ avec $p = 2^{255} - 19$.

1.1.3.4.3 Nombres premiers Mersenne généralisés

Pour ce qui concerne les nombres de Mersenne généralisés, Solinas dans [60] donne des algorithmes repris dans les normes et préconisations du NIST. En effet, le NIST fournit les nombres premiers suivants pour usages cryptographiques dans le cas des protocoles basés sur les courbes elliptiques :

$\rightarrow$ $P192 = 2^{192} - 2^{64} - 1$;
$\rightarrow$ $P224 = 2^{224} - 2^{96} + 1$;
$\rightarrow$ $P256 = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$;
$\rightarrow$ $P384 = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$;
$\rightarrow$ $P521 = 2^{521} - 1$ (déjà mentionné, c'est un nombre de Mersenne.)

L'idée de Solinas est de tirer parti de la forme particulière de ces nombres. Par exemple, il remarque que pour les nombres de la forme $p = 2^{3k} - 2^k + 1$, on peut représenter un entier $A < p^2$ de taille $6k$-bits comme suit :

$$A = \sum_{j=0}^{5} A_j \cdot 2^{jk} \text{ avec } 0 \leq A_j < 2^k.$$

On pose maintenant

$$\sum_{j=0}^{5} A_j \cdot 2^{jk} \equiv \sum_{j=0}^{2} B_j \cdot 2^{jk} \bmod p.$$

Les congruences modulo p suivantes sont valides :

$$\begin{array}{rcll} 2^{3k} & \equiv & -1 + 2^k & \mathrm{mod} p, \\ 2^{4k} & \equiv & -2^k + 2^{2k} & \mathrm{mod} p, \\ 2^{5k} & \equiv & -1 + 2^k - 2^{2k} & \mathrm{mod} p, \end{array}$$

ou encore

$$\begin{array}{rcll} A_3 2^{3k} & \equiv & -A_3 + 2^k A_3 & \mathrm{mod} p, \\ A_4 2^{4k} & \equiv & -2^k A_4 + 2^{2k} A_4 & \mathrm{mod} p, \\ A_5 2^{5k} & \equiv & -A_5 + 2^k A_5 - 2^{2k} A_5 & \mathrm{mod} p. \end{array}$$

On tire de ce qui précède

$$\begin{array}{rcl} B_0 & = & A_0 - A_3 - A_5 \\ B_1 & = & A_1 + A_3 - A_4 + A_5 \\ B_2 & = & A_2 + A_4 - A_5. \end{array}$$

L'entier du NIST $P224$ suit cette forme. Solinas dans [60] fournit des algorithmes analogues pour les autres nombres premiers du NIST.

En conclusion de cette section, nous insistons sur le faible coût de la réduction modulaire dans le cas particulier des modules choisis pour la cryptographie sur courbe elliptique. Les méthodes disponibles sont bien plus rapides que les algorithmes génériques et se résument à un certain nombre de décalages, additions ou soustractions modulaires, ou multiplications faisant intervenir de petites opérandes.

1.1.4 Inversion modulaire

Dans cette section, on considère un élement A non nul du corps $\mathbb{F}_p$. L'inversion d'un élément A de $\mathbb{F}_p$ consiste à calculer son inverse A', tel que :

$$A \cdot A' \equiv 1 \bmod p.$$

En général, A' est noté A^{-1}. Nous présentons ci-après deux algorithmes pour le calcul de l'inverse modulo p : l'algorithme d'Euclide étendu (*Extended Euclidean Algorithm*), et une méthode basée sur le petit théorème de Fermat.

1.1.4.1 Inversion dans $\mathbb{F}_p$ basée sur l'algorithme d'Euclide étendu

Soit A et p deux entiers non nuls. Le plus grand commun diviseur (pgcd) de A et p est le plus grand entier d qui divise à la fois A et p. Le lemme suivant permet de construire des algorithmes efficaces de calcul du pgcd.

Lemme 1.1.2. *Soit A et p deux entiers positifs non nuls. Pour tout entier c, on a*

$$\text{pgcd}(A, p) = \text{pgcd}(p - cA, A).$$

Dans l'algorithme d'Euclide, le pgcd des entiers positifs A et p avec $p \geq A$ est calculé par une suite de divisions. On commence par diviser p par A, et on obtient un quotient q et un reste R tels que $p = qA + R$ avec $0 \leq R < A$. Par le lemme 1.1.2, on a $\text{pgcd}(A, p) = \text{pgcd}(R, A)$. Ceci réduit le problème du calcul du pgcd de A et p par celui du calcul du pgcd de R et A qui sont plus petits que les paramètres initiaux. On répète ces opérations jusqu'à obtenir $\text{pgcd}(0, d) = d$. L'algorithme termine forcément dans la mesure où pour des arguments positifs, la suite des restes est strictement décroissante. Cet algorithme est efficace car le nombre de divisions est au plus deux fois la longueur de la représentation binaire de A (voir [28], chapitre 2, pages 40 et suivantes).

Cet algorithme d'Euclide peut être étendu pour déterminer deux entiers X et Y tels que $AX + pY = \text{pgcd}(A, p)$. Si p est un nombre premier (ou plus généralement, si A et p sont premiers entre eux), on a $\text{pgcd}(A, p) = 1$, et ceci implique que $AX \equiv 1 \bmod p$, et X est l'inverse de A modulo p. C'est ainsi que l'algorithme 1.10 calcule l'inverse de A en calculant le pgcd de A et p. Posons

$$AX_1 + pY_1 = u \text{ et } AX_2 + pY_2 = v \text{ avec } u \leq v.$$

Les valeurs Y_1 et Y_2 ne sont pas utiles, nous ne les calculerons donc pas. On prend pour valeurs initiales $u = A$ et $v = p$. Ceci correspond à $X_1 = 1, X_2 = 0$.

La suite des divisions de v par u permet d'obtenir la suite correspondante des entiers X_1 et X_2. À chaque tour de boucle, on met à jour les valeurs en prenant u pour v et le reste de la division $\lfloor v/u \rfloor$ pour u. On arrête l'algorithme quand $u = 1$, et la dernière valeur de X_1 est l'inverse de A modulo p.

1.1.4.2 Petit théorème de Fermat

Un autre algorithme peut être utilisé. Il tire parti du petit théorème de Fermat (théorème 1.1.1), et calcule l'inverse comme une exponentiation modulaire.

Théorème 1.1.1. *Soit $A, p \in \mathbb{N}$ tels que p est un nombre premier et A n'est pas divisible par p, alors $A^{p-1} - 1$ est un multiple de p, qu'on peut aussi écrire $A^{p-1} - 1 \equiv 0 \mod p$ avec $A \not\equiv 0 \mod p$.*

Algorithme 1.10 Inversion dans $\mathbb{F}_p$ basée sur l'algorithme d'Euclide étendu

Require: $A, p \in \mathbb{N}$, p premier et $0 < A < p$.
Ensure: $A' = A^{-1} \bmod p$
1: $u \leftarrow A, v \leftarrow p, X_1 \leftarrow 1, X_2 \leftarrow 0$
2: **while** $u \neq 1$ **do**
3: $q \leftarrow \lfloor u/v \rfloor, R \leftarrow v - qu, X \leftarrow X_2 - qX_1$
4: $v \leftarrow u, u \leftarrow R, X_2 \leftarrow X_1, X_1 \leftarrow X$
5: **end while**
6: **return** $(X_1 \bmod p)$

La façon la plus efficace de calculer cette exponentiation, c'est de représenter $p-2$ sous la forme d'une chaîne d'additions, et de calculer l'exponentiation avec l'algorithme 1.11.

Une chaîne d'additions est une suite finie d'entiers $V = (v_0, \ldots, v_s)$ et une suite de paires d'entiers $U = ((i_1, j_1), (i_2, j_2), \ldots, (i_s, j_s))$ telles que $v_0 = 1, v_s = p-2$ et $\forall k \leq s, v_k = v_{i_k} + v_{j_k}$ pour $1 \leq k \leq s$ et $0 \leq i_k, j_k < k$.

On peut calculer l'exponentiation à l'aide de la chaîne d'additions. Si $v_s = v_i + v_j = p-2$ avec $i, j < s$, $A^{p-2} = A^{v_i} \times A^{v_j}$. Les variables A^{v_i} et A^{v_j} sont connues de la même manière récursive par les termes précédents de la chaîne d'additions. C'est ainsi que fonctionne l'algorithme 1.11. L'efficacité de cet algorithme tient au fait de disposer de la chaîne d'additions la plus courte pour l'exposant $p-2$. Cette chaîne est calculée une fois pour toute à l'avance, pour chaque corps donné.

Algorithme 1.11 Exponentiation par chaîne d'additions

Require: $A \in \mathbb{F}_p$, une chaîne d'additions représentant $p-2$: $V = \{v_0, \ldots, v_s\}$ et $U = \{(i_1, j_1), (i_2, j_2), \ldots, (i_s, j_s)\}$.
Ensure: $X = A^{-1} \bmod p$
1: initialisation : $X_0 \leftarrow A, X_1 \leftarrow A^2 \bmod p$
2: **for** $k = 2$ to s **do**
3: **if** $i_k = j_k$ **then**
4: $X_k \leftarrow X_{i_k}^2 \bmod p$
5: **else**
6: $X_k \leftarrow X_{i_k} \times X_{j_k} \bmod p$
7: **end if**
8: **end for**
9: **return** X_s

Exemple 1.1.1. *Soit $p = 233$ (p est premier), alors l'inverse d'un élément de $\mathbb{F}_p$ se calcule comme*

$$A^{-1} \equiv A^{231} \bmod p.$$

231 *est représenté en chaîne d'addition comme suit :*

$$\begin{aligned} V &= (1, 2, 3, 4, 7, 14, 28, 56, 112, 224, 231) \\ \text{et } U &= ((0,0), (0,1), (0,2), (2,3), (4,4), (5,5), (6,6), (7,7), (8,8), (4,9)). \end{aligned}$$

Chaque carré effectue un doublement de l'exposant et chaque multiplication de deux puissances ajoute les deux exposants. La table suivante déroule les opérations.

valeur de l'exposant	*opération(s)*	*valeur sauvegardée*
1	*1 carré*	-
2	*1 multiplication par* A	$t = A^3$
3	*1 multiplication par* A	-
4	*1 multiplication par* t	$t = A^7$
$7, 14, 28, 56, 112,$	*5 carrés*	-
224	*1 multiplication par* t	-
231		-
Total	*6 carrés et 4 multiplications*	

La table ci-dessus montre que le coût de cet algorithme dépend en premier lieu de la chaîne d'additions choisie, selon le coût relatif du carré et de la multiplication modulaires. On peut aussi ajouter que cette complexité est constante et ne dépend pas de l'élément à inverser, contrairement à l'algorithme d'Euclide étendu.

Remarque 1.1.1. *Dans le cas d'un anneau $\mathbb{Z}/N\mathbb{Z}$ avec N composite, le théorème d'Euler permet une évaluation de l'inverse quand il existe de façon similaire. On utilise pour cela la fonction $\phi(N)$, l'indicatrice d'Euler. Il s'agit de l'ordre du groupe des éléments inversibles de l'anneau $\mathbb{Z}/N\mathbb{Z}$. Si N est un nombre premier, on a $\phi(N) = N - 1$. Le théorème d'Euler se formule ainsi :*

Théorème 1.1.2. *Pour tout élément inversible A de $\mathbb{Z}/N\mathbb{Z}$, on a $A^{\phi(N)} \equiv 1 \bmod N$.*

1.2 Opérations sur le corps binaire $\mathbb{F}_{2^m}$

1.2.1 Brefs rappels sur le corps $\mathbb{F}_{2^m}$

Le plus petit corps fini comporte deux éléments, il s'agit de $\mathbb{F}_2$. Les éléments sont 0 et 1. On donne ci-après les lois sur ce corps :

Opérations sur $\mathbb{F}_2$			
a	b	$S = a + b$	$S = a \times b$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Le corps $\mathbb{F}_{2^m}$ est une extension du corps $\mathbb{F}_2$. On le construit généralement à l'aide d'un polynôme irréductible $f(x)$ dans $\mathbb{F}_{2^m}$ de degré m. L'ensemble des classes de polynômes modulo $f(x)$ a une structure naturelle de corps. On représente les éléments de ce corps, c'est à dire les classes d'équivalences modulo $f(x)$, par les polynômes à coefficients dans $\mathbb{F}_2$ de $\mathbb{F}_2[x]$ de degré au plus $m - 1$.

Exemple 1.2.1. *Nous montrons ici la structure d'un petit corps, $\mathbb{F}_{2^4}$. Nous posons $f(x)$, le polynôme irréductible à coefficients dans $\mathbb{F}_2$*

$$f(x) = x^4 + x + 1.$$

Soit $g = x \bmod f(x)$. Les éléments de $\mathbb{F}_{2^4}$ sont 0 et les puissances g^i avec $i \in \{0, \ldots, 14\}$, que l'on exprime en polynômes de degré 3 en x, comme dans la table suivante :

élément	*polynôme*	*repr. binaire*	*élément*	*polynôme*	*repr. binaire*
0	0	0000	g^7	x^3+x+1	1011
1	1	0001	g^8	x^2+1	0101
g	x	0010	g^9	x^3+x	1010
g^2	x^2	0100	g^{10}	x^2+x+1	0111
g^3	x^3	1000	g^{11}	x^3+x^2+x	1110
g^4	$x+1$	0011	g^{12}	x^3+x^2+x+1	1111
g^5	x^2+x	0110	g^{13}	x^3+x^2+1	1101
g^6	x^3+x^2	1100	g^{14}	x^3+1	1001

Pour des usages cryptographiques, le NIST dans [19] (pages 90 et suivantes) préconise un certain nombre de corps et le polynôme irréductible correspondant pour chacun. Il s'agit de trinômes ou de pentanômes. Les voici :

— $\mathbb{F}_{2^{163}} : f(x) = x^{163} + x^7 + x^6 + x^3 + 1$;
— $\mathbb{F}_{2^{233}} : f(x) = x^{233} + x^{74} + 1$;
— $\mathbb{F}_{2^{283}} : f(x) = x^{283} + x^{12} + x^7 + x^5 + 1$;
— $\mathbb{F}_{2^{409}} : f(x) = x^{409} + x^{87} + 1$;
— $\mathbb{F}_{2^{571}} : f(x) = x^{571} + x^{10} + x^5 + x^2 + 1$.

La taille de ces corps est équivalente à celle des corps premiers correspondants (voir paragraphe 1.1.3.4.3 page 34), pour usage dans les protocoles basés sur les courbes elliptiques.

1.2.2 Addition

Cette opération, en raison de la caractéristique 2 du corps $\mathbb{F}_{2^m}$, se résume à une opération OU exclusif (XOR) bit à bit. Lorsque l'on additionne deux polynômes $A = \sum_{i=0}^{m-1} a_i \cdot x^i$ et $B = \sum_{i=0}^{m-1} b_i \cdot x^i$, on ne propage pas de retenue comme pour une addition arithmétique classique. En effet :

$$A + B = \sum_{i=0}^{m-1} (a_i + b_i \bmod 2) \cdot x^i.$$

Ainsi, quel que soit le nombre de monômes d'un degré donné que l'on additionne, le coefficient de la somme de ces monômes ne peut prendre que deux valeurs, 0 ou 1.

Nous remarquons aussi qu'additionner ou soustraire revient au même. En effet, la caractéristique 2 implique $2 \equiv 0 \bmod 2$. Autrement dit :

$$\forall A \in \mathbb{F}_{2^m}, \text{ on a : } A + A = A - A = 0.$$

1.2.3 Multiplication

Dans le corps $\mathbb{F}_{2^m}$, la multiplication consiste premièrement en une multiplication des polynômes représentant les éléments du corps, puis en une réduction modulaire. Les polynômes sont de degré $m-1$, et le résultat de la multiplication de polynômes sera de degré $2m-2$. La réduction modulaire (modulo $f(x)$) permettra de représenter le résultat par le polynôme de degré $m-1$ correspondant.

Dans cette section, nous examinons donc en premier la multiplication de polynômes, puis le cas particulier de l'élévation au carré. Nous traitons ensuite de la réduction modulaire.

1.2.3.1 Multiplication de polynômes

On pose :

$$A(x) = a_0 + a_1 x + a_2 x^2 + ... + a_{m-1} x^{m-1} \quad \text{avec } a_i \in \{0, 1\}.$$

Le résultat du produit de deux polynômes A et B de degré $m-1$ est le suivant :

$$A(x) \cdot B(x) = \sum_{i=0}^{m-1} \left(\sum_{k=0}^{i} a_k \cdot b_{i-k} \right) x^i + \sum_{i=m}^{2m-2} \left(\sum_{k=i-m-1}^{m-1} a_k \cdot b_{i-k} \right) x^i.$$

Deux approches sont maintenant présentées. La première est la *Comb Multiplication* qui utilise les opérations classiques (WXOR, WAND, WSHIFT...). La seconde est l'approche de Karatsuba appliquée à la multiplication de polynômes.

1.2.3.1.1 Multiplication dans $\mathbb{F}_{2^m}$ *CombMul*(A,B)

Nous présentons ici un algorithme décrit par Hankerson *et al.* dans [22] et qui utilise une technique de fenêtres avec table (*window technique*). Nous présentons le cas où la taille de la fenêtre est $w = 4$ bits, cette valeur étant retenue dans la littérature comme le meilleur compromis (voir [4, 22]). On représente les polynômes de degré $m-1$ à l'aide de n mots de 64 bits (mais on peut déduire aisément les variantes pour des mots de taille différente, 32 ou 128 bits). Un élément $A = \sum_{i=0}^{m-1} a_i x^i \in \mathbb{F}_2[x]$ est stocké sur $n = \lceil m/64 \rceil$ mots machine. On utilisera dans la suite une décomposition en polynômes de degré au plus 3, soit en paquets de 4 bits comme suit :

$$A = \sum_{i=0}^{\mu} A_i x^{4i}$$

où les polynômes A_i sont de degré < 4 et $\mu = \lceil m/4 \rceil$ est le nombre de paquets de 4 bits.

Soient A et B deux polynômes à multiplier entre eux. On décompose B en polynômes de degré au plus 3 (paquets de 4 bits) comme suit :

$$B = \sum_{j=0}^{n-2} \sum_{k=0}^{15} B_{16j+k} x^{64j+4k} + \sum_{k=0}^{\mu-16(n-1)-1} B_{16(n-1)+k} x^{64(n-1)+4k}$$

où $\deg B_{16j+k} < 4$. Alors, le résultat de la multiplication $C = A \times B$ s'exprime à l'aide de l'expansion de B comme suit

$$\begin{aligned} C &= A \cdot (\textstyle\sum_{j=0}^{n-2} \sum_{k=0}^{15} B_{16j+k} x^{64j+4k} + \sum_{k=0}^{\mu-16(n-1)-1} B_{16(n-1)+4k} x^{64(n-1)+4k}) \\ &= \textstyle\sum_{j=0}^{n-2} \sum_{k=0}^{15} (A \cdot B_{16j+k} x^{64j+4k}) + \sum_{k=0}^{\mu-16(n-1)-1} (A \cdot B_{16(n-1)+k}) x^{64(n-1)+4k} \\ &= \textstyle\sum_{k=0}^{\mu-16(n-1)-1} x^{4k} \left(\sum_{j=0}^{n-1} A \cdot B_{16j+k} x^{64j} \right) + \sum_{k=\mu-16(n-1)}^{15} x^{4k} \left(\sum_{j=0}^{n-2} (A \cdot B_{16j+k} x^{64j}) \right). \end{aligned}$$

Complexité : elle est évaluée en terme d'opérations sur des mots de 64 bits (WXOR, WAND et WShift), et se décompose en trois parties.

— Le calcul de la table T consiste en sept décalages à gauche d'un bit d'un polynôme stocké sur $n = \lceil m/64 \rceil$ mots machine, puis autant de XOR de deux polynômes sur n mots également. Le décalage à gauche est effectué en décalant chaque mot (sauf le dernier) vers la droite de 63 bits pour stocker la retenue à passer au mot suivant, puis à

Algorithme 1.12 Multiplication dans $\mathbb{F}_{2^m}$ *CombMul*(A,B)

Require: $A(x)$ et $B(x)$ polynômes binaires de degré $< 64n$, et $B(x) = \sum_{j=0}^{n-1}\sum_{k=0}^{15} B_{16j+k}x^{4k+64j}$ décomposé en mots de 64 bits et paquets de 4 bits.
Ensure: $C(x) = A(x) \cdot B(x)$

// Calcul préalable de la table $T[u] = u(x) \cdot A(x)$ pour tout u tel que $\deg u(x) < 4$

1: $T[0] \leftarrow 0$
2: $T[1] \leftarrow A$
3: **for** k **from** 1 **to** 7 **do**
4: $\quad T[2k] \leftarrow T[k] << 1$
5: $\quad T[2k+1] \leftarrow T[2k] \oplus A$
6: **end for**

// décalages à gauche et accumulation

7: $C \leftarrow 0$
8: **for** k **from** 15 **downto** 0 **do**
9: $\quad C \leftarrow C << 4$
10: $\quad$ **for** j **from** $n-1$ **downto** 0 **do**
11: $\quad\quad C \leftarrow C \oplus (T[B_{16j+k}] << 64j)$
12: $\quad$ **end for**
13: **end for**
14: **return** (C)

effectuer le décalage d'un bit vers la gauche et il y a donc, pour n mots, $2n-1$ WShift et $n-1$ WXOR pour chaque retenue. Pour le XOR de polynômes, ceci coûte n WXOR par opération.

— Les décalages $C \leftarrow C << 4$ sont effectués de façon similaire. Ainsi, C étant de taille $2n$ mots, chaque décalage de 4 bits $C \leftarrow C << 4$ coûte $4n$ WShift et $2n$ WXOR. Il y en a quinze au total, soient $60n$ WShift et $30n$ WXOR.

— Les accumulations $C \leftarrow C \oplus (T[B_{16j+k}] << 64j)$ sont au nombre de $n \times 16 = \mu$. Il faut donc $\mu - n$ décalages (WShift) et μ masquages (WAND) pour obtenir B_{16j+k}. Ensuite, on effectue μ XOR de polynômes élément de la table T, de taille n, soient $\mu \times n$ WXOR au total.

La complexité totale de l'algorithme 1.12 est fournie dans la table 1.8, qui détaille la complexité de chaque étape.

	Opérations	#WXOR	#WShift	#WAND
7 étapes 4	$T[k] << 1$	$7 \times (n-1)$	$7 \times (2n-1)$	-
7 étapes 5	$T[2k] \oplus A$	$7 \times n$	-	-
15 étapes 9	$C << 4$	$30 \times n$	$60 \times n$	-
$16 \times n$ étapes 11	$T[B_{16j+k}] << 64j$	-	$\mu - n$	μ
	$C \oplus (T[B_{16j+k}] << 64j)$	$n\mu$	-	-
Total		$n\mu + 44n - 7$	$\mu + 73n - 7$	μ

TABLE 1.8 – Complexité de l'algorithme 1.12 (*CombMul*)

De façon plus synthétique, sachant que $n = \lceil m/64 \rceil$ et que $\mu = \lceil m/4 \rceil$, la complexité globale de cet algorithme est donc en $\mathcal{O}(n^2)$ ou $\mathcal{O}(m^2)$.

1.2.3.1.2 Algorithme de Karatsuba

Il s'agit d'un algorithme proposé par Karatsuba dans [53], à l'origine pour effectuer des multiplications sur des grands entiers. Cette méthode sous-quadratique est intéressante si l'on dispose d'une multiplication élémentaire de polynômes efficace, et peut fonctionner de façon récursive. Nous adaptons cet algorithme pour une représentation des polynômes binaires dans $n = 2^s$ mots machine. On suppose que ces mots sont de tailles 64 bits. Cet algorithme se déroule selon les étapes suivantes : soit $A = \sum_{i=0}^{2^s-1} a_i \cdot x^{64i}$ et $B = \sum_{i=0}^{2^s-1} b_i \cdot x^{64i}$ deux polynômes de degré 64×2^s. On note

$$A_0 = \sum_{i=0}^{2^{s-1}-1} a_i \cdot x^{64i}, A_1 = \sum_{i=2^{s-1}}^{2^s-1} a_i \cdot x^{64i}, \text{ et } B_0 = \sum_{i=0}^{2^{s-1}-1} b_i \cdot x^{64i}, B_1 = \sum_{i=2^{s-1}}^{2^s-1} b_i \cdot x^{64i}.$$

— Lors de l'appel de la fonction Karatsuba, on pose :

$$\begin{cases} R_0 = A_0 \times B_0, \\ R_1 = (A_0 + A_1) \times (B_0 + B_1), \\ R_2 = A_1 \times B_1. \end{cases}$$

— On calcule alors successivement, R_0, R_1 et R_2, soit par un appel récursif, soit, si la taille le permet, par l'algorithme de multiplication le plus adapté (multiplication de mots machine) ;
— Enfin, on effectue la reconstruction finale pour obtenir $C = A \times B$, soit :

$$C = R_0 + (R_1 + R_2 + R_0) \cdot x^{64 \times 2^{s-1}} + R_2 \cdot x^{64 \times 2^s}.$$

Algorithme 1.13 Multiplication dans $\mathbb{F}_{2^m}$ *KaratRec*(A,B,n)

Require: A et B représentés par $n = 2^s$ mots machine de 64 bits.
Ensure: $C = A \times B$
  if $n = 1$ **then**
    return ($Mult64(A, B)$)
  else
    // Séparation en deux moitiés de taille $n/2$.
    $A = A_0 + x^{64n/2} A_1$
    $B = B_0 + x^{64n/2} B_1$
    // Appel récursif à la multiplication
    $R_0 \leftarrow$ *KaratRec*$(A_0, B_0, n/2)$
    $R_1 \leftarrow$ *KaratRec*$(A_1, B_1, n/2)$
    $R_2 \leftarrow$ *KaratRec*$(A_0 + A_1, B_0 + B_1, n/2)$
    // Reconstruction
    $C \leftarrow R_0 + (R_0 + R_1 + R_2)X^{64n/2} + R_1 X^{64n}$
    return (C)
  end if

Complexité : En terme de WXOR, une itération des formules de Karatsuba pour des polynômes de taille n mots nécessite n WXORs pour les additions $A_0 + A_1$ et $B_0 + B_1$, et $6n/2$

WXORs pour la reconstruction de R. On obtient la récurrence pour la complexité en partie gauche de (1.3). On fournit en partie droite de (1.3) la forme non récurrente de la complexité.

$$\begin{array}{ll} \left\{ \begin{array}{l} \#WXOR(n)=4n+3\#WXOR(n/2), \\ \#WXOR(1)=0. \end{array} \right. & \Longrightarrow \quad \#WXOR(n) = 8n^{\log_2(3)} - 8n \\ \left\{ \begin{array}{l} \#Mult64(n)=3\#Mult64(n/2), \\ \#Mult64(1)=1. \end{array} \right. & \Longrightarrow \quad \#Mult64(n) = n^{\log_2(3)}. \end{array} \tag{1.3}$$

La complexité de l'algorithme de Karatsuba est inférieure à celle de l'algorithme *CombMul* (voir l'algorithme 1.12), en ce que le nombre d'opérations élémentaires est inférieur. La complexité est maintenant de $\mathcal{O}(n^{\log_2(3)})$, à comparer à la complexité précédente en $\mathcal{O}(n^2)$.

1.2.3.1.3 Élévation au carré

Cette opération est un cas particulier du produit de polynômes. En effet, la caractéristique 2 de $\mathbb{F}_{2^m}$ permet de récrire l'identité remarquable :

$$(A+B)^2 = A^2 + 2AB + B^2 = A^2 + B^2 \ \text{(car } 2 \equiv 0 \bmod 2),$$

Il s'ensuit donc que pour $A(x) = a_0 + a_1 x + a_2 x^2 + ... + a_{m-1}x^{m-1}$, on a :

$$A^2(x) = a_0 + a_1 x^2 + a_2 x^4 + ... + a_{m-1}x^{2(m-1)}.$$

Ainsi, élever un polynôme au carré revient à insérer un 0 entre chacun de ses coefficients binaires.

Exemple 1.2.2. *Pour $A(x) = x^7 + x^4 + x^2 + x + 1$, on a $A = (1,0,0,1,0,1,1,1)$ sur 8 bits , et*

$$A^2(x) = x^{14} + x^8 + x^4 + x^2 + 1.$$

En terme de représentation :

$$A^2 = (\underline{0},1,\underline{0},0,\underline{0},0,\underline{0},1,\underline{0},0,\underline{0},1,\underline{0},1,\underline{0},1) \ \textit{sur 16 bits}\ .$$

Ce calcul pourra se faire d'une façon plus efficace en comparaison d'une multiplication avec deux opérandes identiques. On peut en effet recourir à une table pour simplifier le calcul, car il suffit alors d'additionner au degré convenable les sorties correspondantes de la table. Ce type d'algorithme est décrit par Hankerson *et al.* dans [22], c'est l'algorithme 1.14, ici traitant de polynômes représentés par des mots de 64 bits.

Dans cet algorithme, la table des $T(v)$ comporte ici 256 éléments, soient les carrés de tous les polynômes de degré au plus 7. Cette valeur est retenue par Hankerson *et al.* comme le meilleur compromis. Cette table pourra être initialisée au début de l'exécution d'un code de calcul.

1.2.3.2 Calcul des racines carrées

La caractéristique deux du corps $\mathbb{F}_{2^m}$ offre des propriétés qui rendent efficace le calcul de racines carrées. Nous reprenons la méthode présentée par Fong *et al.* dans [18]. Pour le calcul des racines carrées dans $\mathbb{F}_{2^m}$, Fong *et al.* dans [18] proposent d'exploiter le petit théorème de Fermat (voir le théorème 1.1.1 page 35). Ce théorème se formule de la manière suivante sur le corps $\mathbb{F}_{2^m}$:

$$A^{2^m} = A.$$

Algorithme 1.14 Élévation au carré

Require: A polynôme binaire de degré $m-1$ sur n mots machine de 64 bits.
Ensure: $C(x) = A^2(x)$
1: Calcul préalable : pour chaque octet $v = (v_7, ..., v_1, v_0)$, calculer la valeur sur 16 bits $T(v) = (0, v_7, 0, ..., 0, v_1, 0, v_0)$
2: **for** i from 0 to $n-1$ **do**
3: Soit $A[i] = (u_7, u_6, u_5, u_4, u_3, u_2, u_1, u_0)$ où chaque u_j est un octet
4: $C[2i] \leftarrow (T(u_3), T(u_2), T(u_1), T(u_0))$
5: $C[2i+1] \leftarrow (T(u_7), T(u_6), T(u_5), T(u_4))$
6: **end for**
7: **return** (C)

De là, on tire $\sqrt{A}$ en effectuant $\sqrt{A} = A^{\frac{1}{2}\cdot 2^m} = A^{2^{m-1}}$.

L'élévation au carré dans $\mathbb{F}_{2^m}$ est une opération linéaire. Ceci permet donc d'écrire :

$$\sqrt{A} = \left(\sum_{i=0}^{m-1} a_i x^i\right)^{2^{m-1}} = \sum_{i=0}^{m-1} a_i (x^{2^{m-1}})^i.$$

On peut ensuite séparer A entre ses puissances paires et impaires :

$$\begin{aligned}\sqrt{A} &= \sum_{i=0}^{(m-1)/2} a_{2i}(x^{2^{m-1}})^{2i} + \sum_{i=0}^{(m-3)/2} a_{2i+1}(x^{2^{m-1}})^{2i+1} \\ &= \sum_{i=0}^{(m-1)/2} a_{2i}x^i + \sum_{i=0}^{(m-3)/2} a_{2i+1}x^{2^{m-1}}x^i = \sum_{i \text{ pair}} a_i x^{\frac{i}{2}} + \sqrt{x}\sum_{i \text{ impair}} a_i x^{\frac{i-1}{2}}\end{aligned}$$

Sachant que $\mathbb{F}_{2^m}$ est défini comme $\mathbb{F}_2[x]/(f(x))$ avec m impair, ou $f(x)$ est un trinôme $f(x) = x^m + x^k + 1$, Fong *et al.* remarquent que :

$$\text{pour } k \text{ impair} : \begin{cases} 0 &= x^{m+1} + x^{k+1} + x \text{ (en multipliant par } x) \\ x &= x^{m+1} + x^{k+1} \\ \sqrt{x} &= x^{\frac{m+1}{2}} + x^{\frac{k+1}{2}} \end{cases}$$

$$\text{pour } k \text{ pair} : \begin{cases} x^m &= x^k + 1 \\ x &= x^{-(m-1)}(x^k+1) \text{ (en divisant par } x^{m-1}) \\ \sqrt{x} &= x^{-\frac{m-1}{2}}(x^{\frac{k}{2}}+1) \end{cases}$$

Ceci conduit à un algorithme efficace pour calculer la racine carrée de A. C'est l'algorithme 1.15.

Algorithme 1.15 Calcul de $\sqrt{A}$ dans $\mathbb{F}_{2^m}$

Require: $A = \sum_{i=0}^{m-1} a_i x^i \in \mathbb{F}_{2^m}$, et réduction polynomiale $f(x) = x^m + x^k + 1$.
Ensure: Calcul de $\sqrt{A}$
1: Calcul préalable de $\sqrt{x}$
2: $a_{\text{pair}} \leftarrow (a_{m-1}, \ldots, a_4, a_2, a_0)$ et $a_{\text{impair}} \leftarrow (a_{m-2}, \ldots, a_5, a_3, a_1)$
3: $C \leftarrow \sqrt{x} \cdot a_{\text{impair}} + a_{\text{pair}}$
4: **return** (C)

1.2.3.3 Réduction modulaire dans $\mathbb{F}_{2^m}$

Cette opération est nécessaire à l'issue de multiplications et d'élévations au carré, pour ramener le résultat obtenu de degré au plus $2m-2$ à un polynôme de degré $m-1$.

Effectuer la réduction modulaire d'un polynôme $A(x)$ de degré $\geq m$, c'est chercher le reste $R(x)$ de la division de $A(x)$ par $f(x)$. On a donc :

$$A(x) = Q(x) \cdot f(x) + R(x), \text{ où } \deg(R(x)) < \deg(f(x)) \text{ et } R(x) \equiv A(x) \bmod f(x).$$

Plusieurs articles ([22], [58] et [6]) proposent une façon rapide de faire. $f(x)$ étant connu à l'avance, on peut tirer avantage de sa nature. En effet, le NIST dans [19] préconise des trinômes ou des pentanômes pour $f(x)$. Ces polynômes sont donc extrêmement creux. Plaçons-nous dans le cas où $f(x)$ est un trinôme $f(x) = x^m + x^k + 1$. On remarque que :

$$\begin{cases} x^m & \equiv & x^k + 1 \bmod f(x), \\ x^{2m-k} & \equiv & x^{m-k} + x^k + 1 \bmod f(x). \end{cases}$$

Soit $A(x)$ un polynôme de degré $2m-2$ que l'on souhaite réduire. On peut remplacer les monômes x^k et x^{2m-k} par leur forme réduite modulo $f(x)$ dans l'expression de $A(x)$. Posons :

$$A(x) = A_0(x) + A_1(x) \cdot x^m + A_2(x) \cdot x^{2m-k}$$

$$\text{où } \begin{cases} A_0(x) = \sum_0^{m-1} a_i \cdot x^i, \\ A_1(x) = \sum_m^{2m-k-1} a_i \cdot x^{i-m}, \\ A_2(x) = \sum_{2m-k}^{2m-2} a_i \cdot x^{i-2m+k}, \end{cases}$$

On a alors :

$$R(x) = A(x) \bmod f(x) = A_0(x) + A_1(x) \cdot (x^k + 1) + A_2(x) \cdot (x^{m-k} + x^k + 1).$$

$R(x)$ a bien pour degré au plus $m-1$. Les opérations élémentaires sont uniquement des décalages et des XOR.

1.2.4 Inversion

1.2.4.1 Inversion basée sur l'algorithme d'Euclide étendu

L'algorithme d'Euclide étendu présenté dans le cas des corps premiers $\mathbb{F}_p$ (voir l'algorithme 1.10 page 36) peut s'adapter au cas des corps binaires. Les auteurs de [22] décrivent cette démarche dans l'algorithme 1.16.

1.2.4.2 Petit théorème de Fermat

Cependant, dans le cas des corps de caractéristique 2, le carré et les carrés multiples, comme nous l'avons vu, peuvent être calculés de façon efficace. On préfère donc généralement une démarche basée sur le petit théorème de Fermat, analogue à celle vue plus haut dans le cas des corps premiers.

Cette deuxième méthode est décrite par Rodriguez-Henriquez *et al.* dans [56] en adaptant une proposition d'Itoh et Tsujii dans [27]. Ces auteurs rappellent que le groupe multiplicatif $(\mathbb{F}_{2^m})^*$ est cyclique d'ordre $2^m - 1$, et donc pour tout élément $A \in (\mathbb{F}_{2^m})^*$, on a :

$$A^{-1} \equiv A^{2^m-2} \equiv A^{2(2^{m-1}-1)}.$$

Algorithme 1.16 Inversion algorithme d'Euclide étendu dans $\mathbb{F}_{2^m}$

Require: $A \in \mathbb{F}_{2^m}, A \neq 0$.
Ensure: $A^{-1}(x) \mod f(x)$
1: $u \leftarrow 1, u' \leftarrow 0, c \leftarrow A, d \leftarrow f$
2: **while** $deg(u) \neq 0$ **do**
3: $\quad Q \leftarrow C/d, R \leftarrow d + QC, X \leftarrow u' + Qu$
4: $\quad d \leftarrow C, C \leftarrow R, u' \leftarrow u, u \leftarrow X$
5: **end while**
6: **return** (u)

Ceci est équivalent au petit théorème de Fermat quand le groupe multiplicatif est $\mathbb{F}_p$ avec p premier (voir Section 1.1.4.2 page 35). Dans cette méthode, on réécrit l'exposant

$$2^m - 2 = 2(2^{m-1} - 1) = 2\sum_{j=0}^{m-2} 2^j = \sum_{j=1}^{m-1} 2^j$$

ce qui conduit à

$$A^{2(2^{m-1}-1)} \equiv A^{-1} \equiv \prod_{j=1}^{m-1} A^{2^j}. \tag{1.4}$$

On note dans la suite $\beta_k(A) = A^{2^k-1}$ pour $k \in \mathbb{N}$. De l'équation (1.4), on tire $(\beta_{m-1}(A))^2 \equiv A^{-1}$. On remarque aussi que

$$\beta_{k+j}(A) \equiv \beta_k(A)^{2^j}\beta_j(A). \tag{1.5}$$

En particulier, pour $k = j$, on a :

$$\beta_{2k}(A) \equiv \beta_k(A)^{2^k}\beta_k(A) = \beta_k(A)^{2^k+1}, \tag{1.6}$$

ce qui correspond à une opération de k carrés (carrés multiples) et une multiplication. Ces deux dernières équations permettent d'atteindre le résultat en utilisant de façon récursive une chaîne d'additions descendante pour arriver à $m-1$. Cette chaîne d'additions est définie comme nous l'avons vu plus haut (voir section 1.1.4.2 page 36) par la suite finie $V = (v_0, \ldots, v_s)$ et une suite de paires $U = ((i_1, j_1), (i_2, j_2), \ldots, (i_s, j_s))$ telles que $v_0 = 1, v_s = p - 2$ et $\forall k \leq s, v_k = v_{i_k} + v_{j_k}$ pour $1 \leq k \leq s$. C'est l'algorithme 1.17.

Algorithme 1.17 Inversion de Itoh-Tsuji [56, 27] dans $\mathbb{F}_{2^m}$

Require: $A \in \mathbb{F}_{2^m}, A \neq 0$, une chaîne d'additions U, V de longueur s représentant $m - 1$.
Ensure: $A^{-1}(x) \mod f(x)$
1: $\beta_{v_0} \leftarrow A$
2: **for** $k = 1$ to s **do**
3: $\quad \beta_{v_k} \leftarrow [\beta_{v_{i_k}}(A)]^{2^{v_{j_k}}} \cdot \beta_{v_{j_k}}(A) \bmod f(x)$
4: **end for**
5: **return** $(\beta^2_{v_s}(A) \bmod f(x))$

Enfin, on peut encore améliorer l'algorithme précédent en codant des opérations de *multisquarings* à l'aide de tables calculées au préalable. Cette approche est décrite dans [64]. Les opérations de *multisquarings* se résument alors à des additions d'éléments d'une table.

En terme de complexité, l'algorithme 1.17 consiste en $m-1$ carrés et s multiplications. Nous donnons ci-après un exemple.

Exemple 1.2.3. *Dans le cas* $m = 233$*, on utilise alors* $1 \to 2 \to 3 \to 6 \to 7 \to 14 \to 28 \to 29 \to 58 \to 116 \to 232$*. Nous obtenons alors la suite d'opérations suivante :*

$$\begin{gathered}
C \leftarrow A, \\
C \leftarrow C^2.C = A^{2^2-1}, \\
C \leftarrow C^2.A = A^{2^3-1}, \\
C \leftarrow C^{2^3}.C = A^{2^6-1}, \\
C \leftarrow C^2.A = A^{2^7-1}, \\
C \leftarrow C^{2^7}.C = A^{2^{14}-1}, \\
C \leftarrow C^{2^{14}}.C = A^{2^{28}-1}, \\
C \leftarrow C^2.A = A^{2^{29}-1}, \\
C \leftarrow C^{2^{29}}.C = A^{2^{58}-1}, \\
C \leftarrow C^{2^{58}}.C = A^{2^{116}-1}, \\
C \leftarrow C^{2^{116}}.C = A^{2^{232}-1}, \\
C \leftarrow C^2 = A^{2(2^{232}-1)} = A^{-1}.
\end{gathered}$$

Cet algorithme nécessite 231 carrés de polynômes et 10 produits de polynômes seulement.

Si on ne compte que 3 multisquarings *pour le calcul de* $C^{2^{29}}, C^{2^{58}}, C^{2^{116}}$*, cet algorithme nécessite en tout 3* multisquarings*, 28 élévations au carré et 10 multiplications modulaires dans* $\mathbb{F}_{2^m}$ *seulement.*

En conclusion, cette opération reste la plus coûteuse. La simple évaluation de complexité montre bien que l'inversion coûte plusieurs dizaines de multiplications modulaires dans $\mathbb{F}_{2^m}$.

Chapitre 2

Protocoles, exponentiations modulaires et multiplications scalaires de point de courbe elliptique

Dans ce chapitre, nous poursuivons notre revue de l'état de l'art avec les protocoles de cryptographie asymétrique. Dans la section 2.1, nous présentons les trois principaux en usage : l'échange de clé de Diffie-Hellman, le chiffrement-déchiffrement d'El Gamal, ainsi que la signature. Ces protocoles font usage d'une fonction à sens unique : leur calcul est facile (la complexité est polynomiale) mais le calcul de la réciproque est difficile (idéalement, exponentiel). La sécurité de ces protocoles repose sur ce problème mathématique difficile. Mais la connaissance d'un paramètre rend la résolution de ce problème aisé, c'est ce qu'on appelle la trappe, ou porte dérobée. Nous présentons donc les deux principaux problèmes difficiles sous-jacents à l'élaboration de ces protocoles dans les fonctions à sens unique, la factorisation et le logarithme discret. Nous poursuivons ensuite par l'exposé des principaux algorithmes de calcul des opérations les plus coûteuses dans la mise en œuvre de ces protocoles : l'exponentiation modulaire en section 2.2, et la multiplication scalaire de point de courbe elliptique (sur corps premier de grande caractéristique et de caractéristique 2) en section 2.3.

2.1 Protocoles

2.1.1 Protocoles basés sur le problème de la factorisation

En 1978, Rivest, Shamir et Adleman dans [54] ont proposé pour le chiffrement le protocole RSA. Il s'agit d'un protocole à clé publique. Examinons le chiffrement/déchiffrement. Bob veut envoyer un message à Alice. Pour ce faire, il va avoir besoin d'une clé de chiffrement. Alice génère une paire de clés, l'une publique dont Bob va se servir, mais utilisable par tout le monde, l'autre secrète, connue d'elle seule. Cette clé secrète permet le déchiffrement.

La première étape de géneration de clés est donc effectuée par Alice :

— Alice choisit deux grands nombres premiers p et q ;
— elle calcule $N = p \cdot q$;
— Alice calcule ensuite $\phi(N) = \phi(p) \cdot \phi(q) = (p-1) \cdot (q-1) = N - (p+q-1)$;
— elle choisit maintenant un entier e tel que $1 < e < \phi(N)$ et $pgcd(e, \phi(N)) = 1$;
— enfin, Alice calcule d tel que $d \cdot e \equiv 1 \bmod \phi(N)$.

La fonction $\phi(N)$ est l'indicatrice d'Euler dont nous avons parlé plus haut (voir la section 1.1.4.2 page 37).

$\rightarrow$ e et N sont publiés, c'est la clé publique d'Alice
$\rightarrow$ d est gardé secret, c'est la clé secrète d'Alice.

On note que les nombres premiers p et q, ayant servi à calculer N, doivent aussi être gardés secrets.

Pour chiffrer un message M, Bob va donc se servir de la clé publique d'Alice (e, N) comme suit :

$$C = M^e \bmod N.$$

C est le message chiffré, et Bob peut l'envoyer sans risque à Alice via un canal de communication non sécurisé.

Pour déchiffrer le message de Bob, Alice va utiliser sa clé secrète d comme suit :

$$\begin{aligned} C^d \bmod N &= (M^e)^d \bmod N \\ &= M^{e \cdot d} \bmod N \\ &= M^{e \cdot d \bmod \phi(N)} \bmod N \\ &= M^1 \bmod N \\ &= M \end{aligned}$$

Dans ce protocole, on remarque que l'opération la plus coûteuse est l'exponentiation. En effet, la géneration des clés se fait une fois pour un grand nombre de chiffrements/déchiffrements.

Qu'en est-il de la sécurité de ces communications ? Ève va s'attaquer aux communications entre Alice et Bob. Son but est de retrouver la clé secrète d'Alice, ce qui lui permettra de déchiffrer tous les messages envoyés à Alice.

Au départ, Ève connaît tous les paramètres publics, soient e et N. Elle peut donc chiffrer tous les messages qu'elle veut, et elle aura à sa disposition un ensemble de paires (M, C) en aussi grand nombre qu'elle le souhaite. Cependant, l'avantage que cela lui donne n'est pas significatif en l'état actuel des connaissances. Elle peut alors essayer de déterminer d à l'aide de la relation qu'Alice a utilisé pour la générer, soit $d \cdot e \equiv 1 \bmod \phi(N)$ ou encore $d \equiv e^{-1} \bmod \phi(N)$. Hélas pour Ève, et heureusement pour Alice et Bob, ce calcul nécessite la connaissance de $\phi(N) = \phi(p) \cdot \phi(q) = (p-1) \cdot (q-1) = N - (p+q-1)$, c'est à dire la connaissance de p et q. Ève ne se décourage pas ! Elle va déterminer p et q en factorisant $N = p \cdot q$. Mais la complexité d'un tel problème met Alice et Bob à l'abri des indiscrétions pour un petit moment. En effet, Stinson dans [61] mentionne les algorithmes de factorisation suivants comme étant les plus efficaces :

— le crible quadratique ;
— l'algorithme de factorisation utilisant les courbes elliptiques ;
— le crible algébrique.

En pratique, Stinson dans [61] donne les complexités de ces algorithmes, que nous reproduisons dans la table 2.1.

Crible quadratique	$\mathcal{O}(e^{(1+\mathcal{O}(1))\sqrt{\ln N \ln\ln N}})$
Courbes elliptiques	$\mathcal{O}(e^{(1+\mathcal{O}(1))\sqrt{2\ln p \ln\ln p}})$
Crible algébrique	$\mathcal{O}(e^{(1{,}92+\mathcal{O}(1))(\ln N)^{1/3}(\ln\ln N)^{2/3}})$

TABLE 2.1 – Complexité des algorithmes de factorisation de grands entiers.

La complexité des algorithmes de factorisation est sous-exponentielle. Pour assurer la sécurité, la taille des entiers (ou du plus petit facteur premier dans le cas de l'algorithme utilisant les courbes elliptiques) est convenablement choisie, afin d'avoir une complexité suffisamment grande.

2.1.2 Protocoles basés sur le problème du logarithme discret

De nombreux protocoles sont basés sur ce problème. Nous en examinerons trois parmi les plus utilisés :

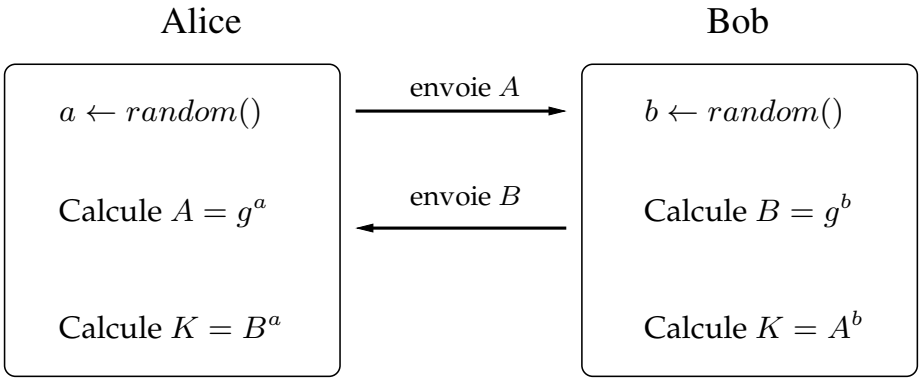
— échange de clés de Diffie-Hellman
Ce protocole proposée par Diffie et Hellman dans [17] en 1976 fonde la cryptographie à clé publique. Il permet à deux partenaires de partager un secret sans le révéler et se l'échanger via un canal de communication non sécurisé.

— chiffrement/déchiffrement El-Gamal
Ce protocole de chiffrement n'est pas sans parenté avec le précédent. Il permet le chiffrement à l'aide d'une clé publique d'un message et son déchiffrement à l'aide d'une clé secrète.

— signature et authentification
Certains de ces protocoles sont définis dans les normes, notamment celles publiées par le *National Institute of Standards and Technology* (NIST), en particulier dans le document *Digital Signature Standard* (DSS) [19] qui définit entre autres le protocole *Digital Signature Algorithm* (DSA) et sa variante *Elliptic Curve Digital Signature Algorithm* (ECDSA). Comme leurs noms l'indiquent, ces protocoles ont pour buts de :
  — garantir l'authenticité : la signature ne doit pas pouvoir être imitée. Autrement dit, si un document porte la signature électronique d'Alice, on doit pouvoir être sûr que c'est effectivement Alice qui l'a signé.
  — garantir la non-répudiation : si Alice a signé un document, elle ne doit pas pouvoir se rétracter et prétendre ensuite qu'elle ne l'a pas signé.
  — garantir l'intégrité : en comparant la signature et le message, on doit pouvoir être sûr que le message n'a pas été altéré lors de l'envoi.

2.1.2.1 Échange de clés Diffie-Hellman

Diffie et Hellman dans [17] ont publié en 1976 un protocole d'échange de clé qui marque un tournant dans la cryptographie moderne. Voici ce processus :



Clé secrète partagée $K = g^{ab}$

Alice et Bob, nos deux partenaires, se sont ici mis d'accord sur le groupe fini $(\mathbb{Z}/p\mathbb{Z})^*$ muni de la multiplication (p est premier). Les opérations effectuées sont des exponentiations. À la fin, Alice et Bob partagent la même clé.

Cependant, n'importe quel groupe peut convenir, sous réserve que le problème du logarithme discret dans ce groupe soit difficile. Ainsi, Victor Miller dans [45] et Neal Koblitz dans [36] ont suggéré indépendamment d'utiliser le groupe des points d'une courbe elliptique définie sur un corps fini. Ceci ne modifie pas fondamentalement ce processus, dont on trouve la description par Silverman *et al.* dans [28], dans la section 5.3.1, pages 292–295. Dans ce cas, Alice et Bob se mettent d'accord au préalable sur une courbe elliptique $E(\mathbb{F}_q)$ et un point de la courbe P générateur de ce groupe.

Pour partager une clé secrète, Alice et Bob vont maintenant utiliser la méthode suivante :

— Alice génère un nombre aléatoire $a \in \mathbb{N}$ et $1 < a < \#E(\mathbb{F}_q)$, calcule la grandeur $a \cdot P$ et l'envoie à Bob ;

— Bob génère un nombre aléatoire $b \in \mathbb{N}$ et $1 < b < \#E(\mathbb{F}_q)$, calcule la grandeur $b \cdot P$ et l'envoie à Alice ;

— Alice calcule alors $a \cdot (b \cdot P)$, et de même, Bob calcule $b \cdot (a \cdot P)$.

Les entiers a et b doivent rester secrets. $\#E(\mathbb{F}_q)$ est l'ordre du groupe formé par l'ensemble des points de la courbe elliptique muni d'une loi d'addition que nous présenterons plus loin. À la fin, Alice et Bob partagent donc la même clé secrète

$$a \cdot (b \cdot P) = b \cdot (a \cdot P).$$

2.1.2.2 Chiffrement El Gamal

Il s'agit d'un protocole de chiffrement à clé publique, proposé par ElGamal dans [62]. Le destinataire met à disposition une clé publique, et tout le monde peut utiliser cette clé pour chiffrer des messages. Proche du protocole d'échange de clé vu précédemment, le protocole de chiffrement El Gamal fonctionne comme suit :

Chiffrement El Gamal		
Clé privée d'Alice	x	
Groupe $\mathcal{G}$	$((\mathbb{Z}/p\mathbb{Z})^*, \times, 1)$	$(E(\mathbb{F}_q), +, \mathcal{O})$
Clé publique d'Alice	$h \leftarrow g^x$ $(\mathcal{G}, g, h)$	$H \leftarrow [x] \cdot P$ $(\mathcal{G}, P, H)$
Chiffrement (Bob) (c_1, c_2)	y = paramètre secret choisi par Bob $c_1 \leftarrow g^y, s \leftarrow h^y$ $c_2 \leftarrow m \cdot s$	$c_1 \leftarrow [y] \cdot P, S \leftarrow [y] \cdot H$ $c_2 \leftarrow m + S$
Déchiffrement (Alice)	$s \leftarrow c_1^x$ $m' \leftarrow c_2 \cdot s^{-1}$ $(m' \leftarrow m \cdot s \cdot s^{-1})$	$S \leftarrow [x] \cdot c_1$ $m' \leftarrow c_2 - S$ $(m' \leftarrow m + S - S)$

Dans ce protocole, s (ou S) est le secret partagé par les parties, et se calcule comme une clé de Diffie-Hellman. En effet, on remarque que

$$s = g^{xy} \text{ ou } S = [xy] \cdot P.$$

Ce protocole est donc essentiellement un échange de clé suivi d'une opération de chiffrement à l'aide de cette clé. Le paramètre y choisi par Bob est à usage unique. Ceci donne la possibilité de produire plusieurs textes chiffrés pour un même message clair et le même destinataire. Ce protocole est souvent choisi pour chiffrer une clé de chiffrement symétrique (crypto-système hybride).

2.1.2.3 Authentification et signature

Nous présentons ici le protocole de signature DSA (*Digital Signature Algorithm*) et sa variante ECDSA (*Elliptic Curve Digital Signature Algorithm*), qui servent à l'authentification. Alice veut signer un message M qu'elle a envoyé à Bob. Elle a envoyé son message M chiffré à l'aide d'un des protocoles précédents par exemple. Bob peut le déchiffrer. Il est donc en possession d'un message M', qu'il a obtenu en déchiffrant ce qu'il a reçu d'Alice. Voici les étapes de ce protocole :

Signature DSA et ECDSA		
Groupe $\mathcal{G}$	$((\mathbb{Z}/p\mathbb{Z})^*, \times, 1)$	$(E(\mathbb{F}_q), +, \mathcal{O})$
Alice hache le message M, en utilisant une fonction de hachage approuvée, et elle obtient $z = \mathcal{H}(M)$;		
Paramètres publics d'Alice	$p \in \mathbb{N}$ premier g un générateur de $\mathcal{G}$ q l'ordre de g	$q \in \mathbb{N}$ premier P un générateur de $\mathcal{G}$ q' l'ordre de P
Clé privée d'Alice	$x \in [1, q-1]$	$x \in [1, q'-1]$
Clé publique d'Alice	$y = g^x \bmod p$	$Q = xP$
Paramètre aléatoire d'Alice tiré au sort à chaque nouvelle signature	$k \in [1, q-1]$	$k \in [1, q'-1]$ $(K_x, K_y) = kP$
Signature (Alice) (r, s) et z	$r = (g^k \bmod p) \bmod q$ $s = (k^{-1}(z + xr)) \bmod q$	$r = K_x \bmod q'$ $s = (k^{-1}(z + xr)) \bmod q'$

Alice envoie donc à (r, s) et z à Bob. On suppose maintenant que M', r' et s' sont les valeurs de M, r et s que Bob a reçues. Bob peut vérifier cette signature en suivant la procédure suivante :

Vérification DSA et ECDSA		
Groupe $\mathcal{G}$	$((\mathbb{Z}/p\mathbb{Z})^*, \times, 1)$	$(E(\mathbb{F}_q), +, \mathcal{O})$
Vérification de la validité de la signature reçue	$0 < r' < q$ et $0 < s' < q$	$0 < r' < q'$ et $0 < s' < q'$
Calcul des grandeurs suivantes	$w = (s')^{-1} \bmod q$ $z = \mathcal{H}(M')$ $u_1 = (zw) \bmod q$ $u_2 = ((r')w) \bmod q$ $u = ((g^{u_1} y^{u_2} \bmod p) \bmod q)$	$w = (s')^{-1} \bmod q'$ $z = \mathcal{H}(M')$ $u_1 = (zw) \bmod q'$ $u_2 = ((r')w) \bmod q'$ $(u, v) = u_1 P + u_2 Q$

Si Bob obtient $u \neq r'$, alors :

— soit le message a été modifié ;

— soit une erreur est apparue dans la génération de clés ;

— ou enfin, un imposteur a tenté de forger une fausse signature.

Dans le cas contraire, le message est authentique, non falsifié et Alice en est bien l'auteur et la signataire. Elle ne pourra se rétracter.

Une preuve de cette vérification peut être trouvée dans [19], annexe E.

2.1.2.4 Logarithme discret

Les opérations effectuées au cours de tous ces protocoles sont maintenant des exponentiations modulo p de nombres entiers ou des multiplications scalaires de points de la courbe elliptique. Ces opérations feront donc l'objet des prochaines sections. Le problème difficile

qu'Ève doit résoudre pour casser ces protocoles consiste à retrouver a, connaissant A, p et g tels que :

$$A = g^a \bmod p.$$

Si r est l'ordre de g, on a $0 \leq a < r$.

Dans le cas de l'exemple précédent de l'échange de clé de Diffie-Hellman, Ève qui écoute les communications ne peut reconstituer la clé, car retrouver a connaissant $A = g^a \bmod p$ ou b connaissant $B = g^b \bmod p$ est un problème difficile, qu'on appelle problème du logarithme discret.

À l'instar du problème de la factorisation, il n'existe pas d'algorithme polynomial permettant de le résoudre.

Les deux algorithmes classiques de résolution de ce problème sont les suivants : l'algorithme *Baby step-Giant step*, dû à Shanks dans [59], et l'algorithme Rho dû à Pollard dans [29]. La complexité de ces algorithmes génériques est $\mathcal{O}(r^{1/2})$, toujours selon Stinson dans [61].

Dans le cas particulier du groupe $\mathbb{Z}_p^*$, Stinson dans [61] mentionne un algorithme particulier. C'est la méthode dite du calcul d'indice. Cet algorithme nécessite un calcul préalable dont le temps de calcul asymptotique est

$$\mathcal{O}\left(e^{(1+\mathcal{O}(1))\sqrt{\ln p \ln\ln p}}\right)$$

et le temps de calcul de la recherche d'un logarithme discret particulier est

$$\mathcal{O}\left(e^{(1/2+\mathcal{O}(1))\sqrt{\ln p \ln\ln p}}\right).$$

Cet algorithme n'est cependant pas générique, puisque son usage est limité au groupe $\mathbb{Z}_p^*$.

2.2 Exponentiation modulaire

2.2.1 Principaux algorithmes d'exponentiation

Nous avons vu au chapitre premier comment effectuer des multiplications et des carrés modulaires, nous pouvons nous pencher à présent sur le problème qui nous intéresse : connaissant g et e, dans un anneau ou un corps $\mathbb{Z}/N\mathbb{Z}$, on veut connaître

$$X = g^e \bmod N = \overbrace{(\ldots(g \times g \bmod N)\ldots \times g \bmod N)}^{e \text{ fois}}$$

L'entier e étant grand, il faut disposer d'un algorithme logarithmique en e, ou polynomial en sa taille. Autrement, tous nos protocoles seraient impraticables...

La méthode la plus directe, et l'une des plus efficaces, c'est d'utiliser la représentation de e. Dans nos ordinateurs, cette représentation est le plus souvent en base 2. La voici :

$$e = \sum_{i=0}^{t-1} e_i 2^i \text{ avec } e_i \in \{0,1\}.$$

Ceci conduit à l'écriture suivante :

$$X = g^{\sum_{i=0}^{t-1} e_i 2^i} \bmod N = \left(\prod_{i=0}^{t-1} (g^{2^i} \bmod N)^{e_i}\right) \bmod N \text{ avec } e_i \in \{0,1\}.$$

Les valeurs $\left(g^{2^i} \bmod N\right)$ sont les carrés modulaires successifs de g, et nous savons les calculer efficacement. Quant aux multiplications modulaires, nous ne les effectuerons que dans le cas $e_i = 1$. Ceci donne corps à un premier algorithme appelé *Square-and-multiply*.

Il existe en deux versions, *Left-to-right* et *Right-to-left*, selon le sens de parcours des bits de la représentation binaire de e. Ils sont décrits dans les algorithmes 2.1 et 2.2.

Algorithme 2.1 *Left-to-right Square-and-multiply*

Require: Les entiers g et $e = (e_{t-1}, \dots e_0)_2 \in [1, \dots, N[$, avec $e_{t-1} = 1$.
Ensure: $X = g^e \bmod N$
1: $X \leftarrow g$
2: **for** $i = t - 2$ downto 0 **do**
3: $\quad X \leftarrow X^2 \bmod N$
4: $\quad$ **if** $e_i = 1$ **then**
5: $\quad\quad X \leftarrow X \cdot g \bmod N$
6: $\quad$ **end if**
7: **end for**
8: **return** $(X = g^e)$

Algorithme 2.2 *Right-to-left Square-and-multiply*

Require: Les entiers g et $e = (e_{t-1}, \dots e_0)_2 \in [1, \dots, N[$, avec $e_{t-1} = 1$.
Ensure: $X = g^e \bmod N$
1: $X \leftarrow 1$
2: **for** $i = 0$ to $t - 1$ **do**
3: $\quad$ **if** $e_i = 1$ **then**
4: $\quad\quad X \leftarrow X \cdot g \bmod N$
5: $\quad$ **end if**
6: $\quad g \leftarrow g^2 \bmod N$
7: **end for**
8: **return** $(X = g^e)$

On remarque que la complexité de ces deux algorithmes est la même : pour un exposant e pris aléatoirement, le nombre d'élévations au carré est de t et le nombre de multiplications est de $t/2$ en moyenne.

On trouve dans la littérature de nombreuses variantes de ces deux algorithmes. Nous aurons l'occasion d'en détailler plus loin dans notre travail. D'une façon générale, ces variantes poursuivent un but précis, souvent l'amélioration des performances, en tirant avantage de particularités du protocole, de possibilités ou non de mémorisation. Un autre but recherché dans la littérature, c'est la résistance à certaines attaques. Dans la partie concernant nos contributions, nous poursuivrons nous aussi ces mêmes objectifs.

2.3 Multiplication de point de courbe elliptique

Dans cette section, nous nous attachons à présenter l'essentiel concernant la multiplication de point de courbe elliptique. Cette partie sera plus étoffée que celle concernant l'exponentiation modulaire. En effet, nous examinons deux cas différents de courbes : les courbes sur corps fini premier de grande caractéristique $\mathbb{F}_p$, et le cas des courbes sur corps de caractéristique

deux $\mathbb{F}_{2^m}$. De plus, comme nous allons le montrer en préambule, l'arithmétique des courbes elliptiques présente une richesse plus importante.

Dans cette section, nous rappelons en premier lieu les généralités sur les courbes elliptiques, puis dans les deux sous-sections suivantes, nous examinons le cas des courbes sur corps premiers, puis binaires.

2.3.1 Qu'est une courbe elliptique ?

Nous nous inspirons ici de la présentation faite par Hankerson *et al.* dans [23].

Définition 2.3.1. *Une courbe elliptique sur un corps $\mathcal{K}$ est définie par une équation de la forme*

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{2.1}$$

où $a_1, a_2, a_3, a_4, a_6 \in \mathcal{K}$ et $\Delta \neq 0$. Δ est le discriminant de E et est défini comme suit :

$$\begin{array}{rcl} \Delta & = & -d_2^2d_8 - 8d_4^3 - 27d_6^2 + 9d_2d_4d_6, \\ d_2 & = & a_1^2 + 4a_2, \\ d_4 & = & 2a_4 + a_1a_3, \\ d_6 & = & a_3^2 + 4a_6, \\ d_8 & = & a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2. \end{array} \tag{2.2}$$

Remarque 2.3.1. *Quelques commentaires sur cette définition :*

(i) L'equation (2.1) est appelée équation de Weierstrass.

(ii) La condition $\Delta \neq 0$ assure que la courbe est lisse (on peut définir une tangente en tout point).

Les courbes elliptiques peuvent prendre différentes morphologies selon leur équation. Nous donnons dans la figure 2.1 deux exemples de courbes sur le corps $\mathbb{R}$.

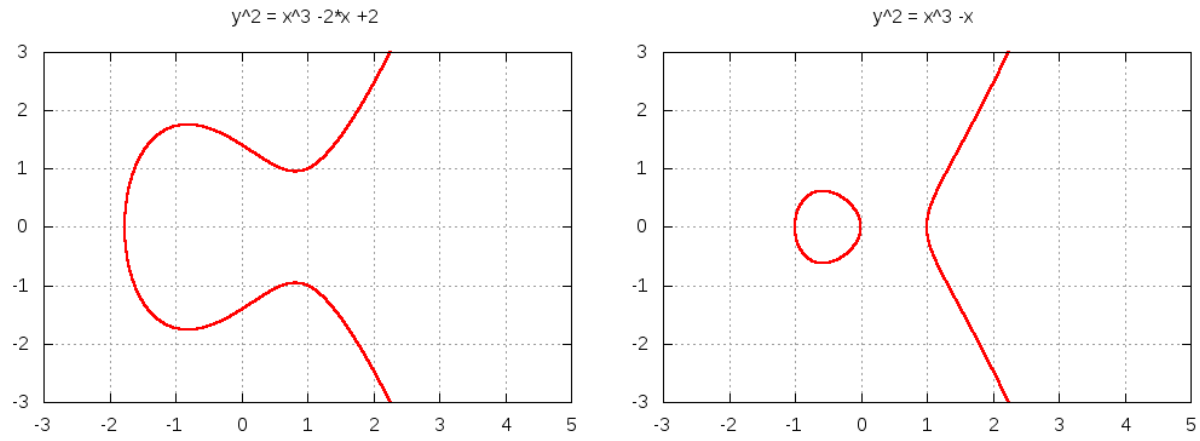


FIGURE 2.1 – Courbes elliptiques sur $\mathbb{R}$.

D'une façon courante, on utilise une forme simplifiée de l'équation de Weierstrass. En effet, Hankerson *et al.* dans [23] font remarquer que deux courbes E_1 et E_2 sont isomorphes entre elles si l'on peut trouver $u, r, s, t \in \mathcal{K}, u \neq 0$, tels que le changement de variable

$$(x, y) \rightarrow (u^2x + r, u^3y + u^2sx + t)$$

transforme l'équation de E_1 en l'équation de E_2. Ceci permet d'obtenir des formes courtes de l'équation de Weierstrass. Nous donnons la démarche dans les deux cas qui nous intéressent, soit celui des courbes sur $\mathbb{F}_p$, puis celui des courbes sur $\mathbb{F}_{2^m}$.

2.3.1.1 Courbe sur un corps $\mathbb{F}_p$ de grande caractéristique $\neq 2, 3$

Nous partons de l'équation générale de Weierstrass (équation 2.1) :

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

Le premier membre peut être récrit de la façon suivante :

$$\begin{aligned} y^2 + a_1xy + a_3y &= (y + \tfrac{a_1x+a_3}{2})^2 - (\tfrac{a_1x+a_3}{2})^2 \\ &= Y^2 - \tfrac{a_1}{4}x^2 - \tfrac{a_1a_3}{2}x - \tfrac{a_3^2}{4} \end{aligned}$$

en posant $Y = (y + \frac{a_1x+a_3}{2})$. L'équation (2.1) peut maintenant se récrire de la façon suivante :

$$E : Y^2 = x^3 + \frac{4a_2 + a_1^2}{4}x^2 + \frac{2a_4 + a_1a_3}{2}x + a_6 + \frac{a_3^2}{4}$$

On traite le second membre de la même manière :

$$x^3 + \frac{4a_2 + a_1^2}{4}x^2 + \frac{2a_4 + a_1a_3}{2}x + a_6 + \frac{a_3^2}{4} \;=\; X^3 + \left[\beta - 3\alpha^2\right] X + \left[\gamma + 2\alpha^3 - \alpha\beta\right]$$

avec $\alpha = \frac{4a_2+a_1^2}{12}, \beta = \frac{2a_4+a_1a_3}{2}, \gamma = a_6 + \frac{a_3^2}{4}$ et $X = (x + \alpha)$.
En posant pour finir $a = \left[\beta - 3\alpha^2\right]$ et $b = \left[\gamma + 2\alpha^3 - \alpha\beta\right]$ avec $a, b \in \mathbb{F}_p$, nous obtenons maintenant la forme courte de l'équation de Weierstrass pour les courbes elliptiques sur $\mathbb{F}_p$:

$$E : Y^2 = X^3 + aX + b \tag{2.3}$$

Toute courbe elliptique sur $\mathbb{F}_p$ est isomorphe par un tel changement de variable à une courbe dont l'équation est sous la forme courte que nous venons de présenter.

2.3.1.2 Courbe sur un corps $\mathbb{F}_{2^m}$

On procède de manière similaire. Deux cas se présentent :

— $a_1 \neq 0$: la courbe est dite alors non super-singulière. Par changement de variable, on trouve la forme réduite suivante :

$$E : Y^2 + XY = X^3 + aX^2 + b \tag{2.4}$$

— $a_1 = 0$: la courbe est dite alors super-singulière. Par changement de variable, on trouve la forme réduite suivante :

$$E : Y^2 + cY = X^3 + aX + b \tag{2.5}$$

Dans ce dernier cas, le changement de variable est $(X, Y) \rightarrow (x + a_2, y)$ et $c = a_3$.

2.3.1.3 Synthèse

Les changements de variables sont fournis par Hankerson *et al.* dans [23]. Nous donnons les formes courtes ou simplifiées dans les deux cas dans la table suivante :

Équation simplifiée de Weierstrass

$E(\mathbb{F}_p)$	$E(\mathbb{F}_{2^m})$ (courbe non super-singulière)
$y^2 = x^3 + ax + b$ $a, b \in \mathbb{F}_p$	$y^2 + xy = x^3 + ax^2 + b$ $a, b \in \mathbb{F}_{2^m}$

2.3.1.4 Loi de groupe

Nous voulons disposer d'un groupe pour y effectuer nos multiplications scalaires. L'ensemble des points de notre courbe elliptique doit donc être muni d'une loi d'addition associative et commutative, et disposer d'un élément neutre.

Reprenons notre exemple sur $\mathbb{R}$. Nous donnons ici la classique loi corde-tangente sous forme graphique dans la figure 2.2. On souhaite additionner deux points de la courbe P et Q. On trace la corde qui les relie. Comme la courbe elliptique est de degré 3, cette droite la coupe en un troisième point. On trace la verticale passant par ce point, on obtient un quatrième point. On baptise ce point $P+Q$, il est la somme de deux premiers. Cette loi d'addition est associative et commutative.

Pour le doublement de points, le tracé de la corde est remplacé par la tangente. Ceci est aussi montré dans la figure 2.2.

L'élément neutre est constitué par le point à l'infini noté $\mathcal{O}$. En effet, la corde qui relie $\mathcal{O}$ avec un point quelconque de la courbe est verticale. Par construction, la somme d'un point P avec le point à l'infini est donc P lui-même.

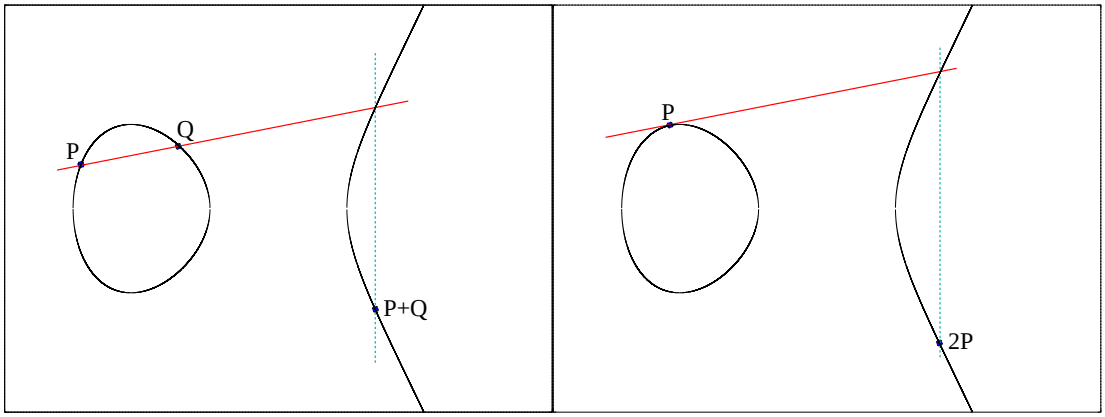


FIGURE 2.2 – Courbes elliptiques sur $\mathbb{R}$, addition de points à gauche, doublement de point à droite.

Cette loi d'addition est dénommée corde-tangente, de par le principe de la construction graphique.

Maintenant, l'ensemble des points de la courbe elliptique, avec le point à l'infini $\mathcal{O}$, muni de cette loi d'addition, forme un groupe commutatif. Nous pouvons donc définir la multiplication scalaire de points de courbe elliptique comme suit : soit $k \in \mathbb{N}$ et $P \in E$, on a

$$k \cdot P = \overbrace{P + P + \ldots + P}^{k \text{ fois}}.$$

Dans la suite de cette section sur les courbes elliptiques, nous examinons dans un premier temps le cas des courbes elliptiques sur corps fini en grande caractéristique, puis dans un deuxième temps, le cas des courbes elliptiques sur corps fini en caractéristique deux, en détaillant les spécificités des opérations sur le corps et l'arithmétique de la courbe dans les deux cas. Nous donnerons dans chaque cas les formules analytiques explicites d'addition et de doublement correspondant à la loi graphique corde-tangente.

Remarque 2.3.2. *Pour les applications cryptographiques et selon le niveau de sécurité, l'ordre du groupe doit être grand. Ceci est assuré par la taille du corps et le choix des coefficients a et b de l'équation*

courte de Weierstrass. L'ordre du groupe doit donc s'écrire $\#E(\mathbb{F}_q) = h \times r$ où h est un petit entier appelé le cofacteur et r un grand entier premier. Le niveau de sécurité correspond à la complexité du problème du logarithme discret dans le sous-groupe d'ordre r. D'après le théorème de Hass, dont on trouve une présentation par Hankerson et al. *dans [23], la relation entre r et q est la suivante :*

$$q + 1 - \sqrt{q} \leq h \times r \leq q + 1 + \sqrt{q},$$

ce qui indique que les tailles de q et r sont du même ordre de grandeur.

En corollaire de cette contrainte, on observe que les coordonnées des points utilisés dans les calculs sont des éléments du corps $\mathbb{F}_q$, dont la taille est celle de q.

Par exemple, si on souhaite un niveau de sécurité de 128 bits, le problème du logarithme discret de la complexité correspondante correspondra à une courbe elliptique munie d'un sous-groupe d'ordre 256 bits, sur un corps d'une taille au moins égale à 256 bits.

2.3.2 Opérations sur le groupe des points de la courbe elliptique sur corps fini en grande caractéristique $\mathbb{F}_p$

On considère une courbe elliptique définie par une forme courte de Weierstrass. Un point $P = (x, y)$ appartient à la courbe elliptique choisie si $(x, y) \in \mathbb{F}_p^2$ satisfont l'équation de la courbe.

Hankerson *et al.* dans [23] fournissent un théorème concernant les classes d'isomorphismes de courbes elliptiques qui permet par changement de variable, et de façon analogue à ce que nous avons vu précédemment pour obtenir la forme courte, de transformer la forme courte en une autre ayant une valeur de a qui permet d'optimiser l'efficacité des formules de doublement et d'addition de points. Les courbes elliptiques recommandées par le NIST ont toutes $a = -3$. Nous nous placerons dans ce cas dans la suite de cet exposé. Nous invitons le lecteur à se reporter à [23] (théorème 3.15 page 84 et paragraphe 3.22 page 89) pour plus de détails.

Dans cette section, notre objectif est de présenter comment la loi de groupe présentée plus haut se matérialise dans le cas d'un corps fini. En effet, la loi d'addition et de doublement corde-tangente doit maintenant être traduite en formulation algébrique, dans le cas du corps fini $\mathbb{F}_p$ qui nous intéresse. Nous présenterons donc l'addition et le doublement en coordonnées affines, puis le système de coordonnées jacobiennes et enfin, nous conclurons par un bilan de complexité.

2.3.2.1 Addition et doublement en coordonnées affines

Ces deux opérations s'effectuent entre points d'une courbe elliptique, et le résultat est un autre point de la courbe elliptique. Pour plus de détails, le lecteur pourra se reporter aux livre et article suivant, dont nous tirons les formules ci-après : [28] [1] et [12]. Nous donnons maintenant la formulation algébrique de la loi corde-tangente. L'addition entre deux points d'une courbe elliptique P_1 et P_2 s'écrit :

$$P_3 = P_1 + P_2 \text{ avec } \begin{cases} P_1 = (x_1, y_1), \\ P_2 = (x_2, y_2), \\ P_3 = (x_3, y_3), \end{cases} \quad \text{où : } \begin{cases} x_3 = \lambda^2 - x_1 - x_2, \\ y_3 = \lambda(x_1 - x_3) - y_1, \end{cases}$$

et :

$$\begin{cases} \lambda = \frac{y_2 - y_1}{x_2 - x_1} \text{ si } P_1 \neq P_2, \\ \lambda = \frac{3x_1^2 + a}{2y_1} + x_1 \text{ si } P_1 = P_2. \end{cases}$$

On ajoute pour finir que le point $-P$, tel que $P + (-P) = \mathcal{O}$, a pour coordonnées $(x, -y)$.

1. Le chapitre 5, intitulé *Elliptic Curves and Cryptography*, traite exclusivement des courbes elliptiques.

2.3.2.2 Coordonnées jacobiennes

Nous avons vu dans le chapitre précédent que l'inversion est l'opération la plus coûteuse. Il est donc utile d'utiliser des systèmes de coordonnées projectives, tel que le système de coordonnées jacobiennes.

Ces systèmes de coordonnées viennent de la remarque suivante (voir Hankerson *et al.* dans [23]) : sur $\mathbb{F}_p$, en posant c et d deux entiers positifs, on peut définir une relation d'équivalence $\simeq$ sur $\mathbb{F}_p^3 \backslash \{(0,0,0)\}$ des triplets non-nuls sur $\mathbb{F}_p$ par

$$(X_1, Y_1, Z_1) \simeq (X_2, Y_2, Z_2) \text{ si } X_1 = \lambda^c X_2, Y_1 = \lambda^d Y_2, Z_1 = \lambda Z_2 \text{ pour } \lambda \in \mathbb{F}_p$$

Le système de coordonnées jacobiennes correspond à $c = 2$ et $d = 3$. Brown *et al.* dans [12] le présentent sous la forme suivante :

$$P = (x, y) \text{ en } (X : Y : Z), \text{ avec } \begin{cases} x = \frac{X}{Z^2}, \\ \\ y = \frac{Y}{Z^3}. \end{cases}$$

Si on remplace x par $\frac{X}{Z^2}$ et y par $\frac{Y}{Z^3}$ dans l'équation courte de Weierstrass et dans les formules de doublement en coordonnées affines données plus haut, on obtient la forme projective de l'équation de Weierstrass :

$$Y^2 = X^3 + aXZ^4 + bZ^6.$$

Pour $P = (X_1, Y_1, Z_1) \in E$ et $P \neq -P$, on a $P = (X_1/Z_1^2, Y_1/Z_1^3, 1)$, et en posant $2P = (X_3', Y_3', 1)$, on a

$$X_3' = \left(\frac{3\frac{X_1^2}{Z_1^4} + a}{2\frac{Y_1}{Z_1^3}} \right)^2 - 2\frac{X_1}{Z_1^2} = \frac{(3X_1^2 + aZ_1^4)^2 - 8X_1Y_1^2}{4Y_1^2Z_1^2}$$

et

$$Y_3' = \left(\frac{3\frac{X_1^2}{Z_1^4} + a}{2\frac{Y_1}{Z_1^3}} \right) \left(\frac{X_1}{Z_1^2} - X_3' \right) - \frac{Y_1}{Z_1^3} = \frac{\left(3X_1^2 + aZ_1^4\right)\left(4X_1Y_1^2 - 4X_3'Y_1^2Z_1^2\right) - 8Y_1^4}{8Y_1^3Z_1^3}.$$

On peut maintenant éliminer les dénominateurs dans les expressions de X_3' et Y_3' en posant $Z_3 = 2Y_1Z_1$ et $X_3 = X_3' \times Z_3^2$, $Y_3 = Y_3' \times Z_3^3$. On obtient alors les formules suivantes pour $2P = (X_3, Y_3, Z_3)$:

$$\begin{array}{lcl} X_3 & = & (3X_1^2 + aZ_1^4) - 8X_1Y_1^2 \\ Y_3 & = & (3X_1^2 + aZ_1^4)(4X_1Y_1^2 - X_3) - 8Y_1^4 \\ Z_3 & = & 2Y_1Z_1 \end{array} \tag{2.6}$$

Pour optimiser le nombre de multiplications modulaires et d'élévations au carré, on peut utiliser des variables pour stocker des résultats intermédiaires. Ceci conduit aux formules données dans la table 2.2.

Une démarche similaire conduit à des formules pour l'addition. Dans le cas où l'un des points est en coordonnées jacobiennes et l'autre point en coordonnées affines (avec $Z_2 = 1$ dans ce cas), on obtient une addition en coordonnées mixées. Les formules correspondantes sont fournies dans la table 2.3.

L'intérêt des systèmes de coordonnées projectives réside dans le fait de supprimer l'opération d'inversion modulaire dans les formules d'addition et de doublement de points.

$$2(X_1 : Y_1 : Z_1) = (X_3 : Y_3 : Z_3) \text{ avec}$$

$$\begin{cases} A = 4X_1 \cdot Y_1^2, B = 8Y_1^2, C = 3(X_1 - Z_1^2) \cdot (X_1 + Z_1^2), D = -2A + C^2, \\ X_3 = D, Y_3 = C \cdot (A - D), Z_3 = 2Y_1 \cdot Z_1. \end{cases}$$

TABLE 2.2 – Formules pour le doublement en coordonnées jacobiennes, courbe $E(\mathbb{F}_p)$ d'après les auteurs dans [12].

$$(X_1 : Y_1 : Z_1) + (X_2 : Y_2 : 1) = (X_3 : Y_3 : Z_3),$$

$$\begin{cases} A = X_2 \cdot Z_1^2, B = Y_2 \cdot Z_1^3, C = A - X_1, D = B - Y_1, \\ X_3 = D^2 - (C^3 + 2X_1 \cdot C^2), Y_3 = D \cdot (X_1 \cdot C^2 - X_3) - Y_1 \cdot C^3, Z_3 = Z_1 \cdot C. \end{cases}$$

TABLE 2.3 – Formules pour l'addition en coordonnées mixées (jacobiennes et affines), courbe $E(\mathbb{F}_p)$, d'après les auteurs dans [12].

Remarque 2.3.3. *On pose $P = (X, Y, Z)$. On peut représenter le point $-P$ en coordonnées jacobiennes :*

$$-P = (X, -Y, Z).$$

Les coordonnées jacobiennes permettent aussi de représenter l'élément neutre ou point à l'infini :

$$\mathcal{O} = (1, 1, 0).$$

2.3.2.3 Opérations pour la courbe Jacobi Quartic

Les courbes elliptiques peuvent être définies par d'autres types d'équation que l'équation de Weierstrass. Dans la littérature, on trouve notamment les courbes Hessiennes, d'Edwards, etc. Nous utilisons dans notre travail la courbe que nous présentons maintenant, la courbe Jacobi Quartic.

Cette courbe a été suggérée par Billet et Joye dans [10]. L'équation de la courbe est

$$y^2 = x^4 - \frac{3}{2}\theta x^2 + 1, \theta \in \mathbb{F}_p. \tag{2.7}$$

La courbe Jacobi Quartic (2.7) est isomorphe à la courbe de Weierstrass suivante :

$$y^2 = x^3 + ax + b \text{ où } a = (-16 - 3\theta^2)/4 \text{ et } b = -\theta^3 - a\theta.$$

Pour une telle courbe, les formules d'addition et de doublement sont unifiées. Soient $P_1 = (x_1, y_1), P_2 = (x_2, y_2), P_3 = (x_3, y_3)$, trois points de E tels que $P_3 = P_1 + P_2$, on a alors :

$$\begin{cases} x_3 = (x_1y_2 + y_1x_2)/(1 - (x_1x_2)^2), \\ y_3 = ((1 + (x_1x_2)^2)(y_1y_2 + 2ax_1x_2) + 2x_1x_2(x_1^2 + x_2^2))/(1 - (x_1x_2)^2)^2. \end{cases}$$

Nous avons présenté le système de coordonnées jacobiennes qui évite les inversions modulaires dans la section précédente. Dans le cas de la courbe Jacobi Quartic, on utilise plutôt le système de coordonnées $XXYZZ$ qui est comme suit :

$$P = (x, y) \text{ en } (X : XX : Y : Z : ZZ), \text{ avec } \begin{cases} x &= X/Z, \\ y &= Y/ZZ, \\ XX &= X^2, \\ ZZ &= Z^2. \end{cases}$$

Pour les opérations de doublement et d'addition de points, les formules suivantes ont été proposées par Hisil *et al.* dans [25] et [26]. Elles sont reprises dans [3]. Elles sont fournies respectivement dans la table 2.4, la table 2.5 et la table 2.6.

$$(X_3, XX_3, Y_3, Z_3, ZZ_3) = 2 \cdot (X_1, XX_1, Y_1, Z_1, ZZ_1).$$

$$\begin{cases} B = XX_1 - ZZ_1, T_1 = XX_1 + ZZ_1, C = Y_1T_1 \\ X_3 = C - Y_1(X_1 + Z_1)^2, Z_3 = T_1B, XX_3 = X_3^2 \\ ZZ_3 = Z_3^2, T_3 = XX_3 + ZZ_3, Y_3 = 2C^2 - T_3 \end{cases}$$

TABLE 2.4 – Formules pour le doublement en coordonnées $XXYZZ$, courbe Jacobi Quartic sur $\mathbb{F}_p$.

$$(X_3, XX_3, Y_3, Z_3, ZZ_3) = (X_1, XX_1, Y_1, Z_1, ZZ_1) + (X_2, XX_2, Y_2, 1, 1).$$

$$\begin{cases} R_1 = (X_1 + Z_1)^2 - XX_1 - ZZ_1, R_2 = 2X_2, A = 2XX_1XX_2, B = 2ZZ_1 \\ C = R_1R_2, D = Y_1Y_2, X_3 = (R_1 + Y_1)(R_2 + Y_2) - C - D \\ Z_3 = B - A, XX_3 = X_3^2, ZZ_3 = Z_3^2, F = A + B + C \\ G = 2((XX_1 + ZZ_1)(XX_2 + 1) + D) + kC, H = XX_3 + ZZ_3, Y_3 = FG - H \end{cases}$$

TABLE 2.5 – Formules pour l'addition en coordonnées mixées affines et $XXYZZ$, courbe Jacobi Quartic sur $\mathbb{F}_p$.

2.3.2.4 Bilan de complexité

Nous donnons dans la table 2.7 le coût des formules de doublements et d'additions en nombre d'inversions, multiplications, carrés et réductions modulaires. Ce bilan montre le moindre coût du doublement de point. Dans le cas de l'addition avec coordonnées mixées, le fait d'avoir un point en coordonnées affines comme opérande permet d'économiser des multiplications.

On remarque que la complexité des opérations pour la courbe Jacobi Quartic est inférieure à celle des opérations sur la courbe de Weierstrass. De plus, d'après [3], la courbe Jacobi Quartic est celle pour laquelle les complexités sont les meilleures pour les opérations sur points parmi tous les différents types de courbes recensés.

$$(X_3, XX_3, Y_3, Z_3, ZZ_3) = (X_1, XX_1, Y_1, Z_1, ZZ_1) + (X_2, XX_2, Y_2, Z_2, ZZ_2).$$

$$\begin{cases} R_1 = (X_1 + Z_1)^2 - XX_1 - ZZ_1, R_2 = (X_2 + Z_2)^2 - XX_2 - ZZ_2 \\ A = 2XX_1XX_2, B = 2ZZ_1ZZ_2, C = R_1R_2, D = Y_1Y_2 \\ X_3 = (R_1 + Y_1)(R_2 + Y_2) - C - D, Z_3 = B - A, XX_3 = X_3^2, ZZ_3 = Z_3^2 \\ F = A + B + C, G = 2((XX_1 + ZZ_1)(XX_2 + ZZ_2) + D) + kC \\ H = XX_3 + ZZ_3, Y_3 = FG - H \end{cases}$$

TABLE 2.6 – Formules pour l'addition en coordonnées $XXYZZ$, courbe Jacobi Quartic sur $\mathbb{F}_p$.

Complexités pour les opérations de points	Equation de Weierstrass avec coord. affines	Equation de Weierstrass avec coord. jacobiennes	Courbe Jacobi Quartic avec coord. $XXYZZ$
Doublement	$2M + 2S + 4R + 1I$	$4M + 4S + 8R$	$3M + 4S + 7R$
Addition	$2M + 1S + 3R + 1I$	$13M + 2S + 15R$	$7M + 4S + 11R$
Addition en coord. mixées	-	$9M + 3S + 12R$	$6M + 3S + 9R$

M = multiplications, S = carrés, R = réductions modulaires, I = inversions modulaires.

TABLE 2.7 – Complexité des opérations de points de courbe elliptique sur $\mathbb{F}_p$.

2.3.3 Opérations sur le groupe des points de la courbe elliptique sur corps fini en caractéristique 2

Les complexités des opérations sur le corps $\mathbb{F}_{2^m}$ (voir la section 1.2 pages 37 et suivantes) diffèrent de celle des opérations sur corps premier. Les algorithmes comme les coûts relatifs entre opérations sont différents. Il en est de même pour les opérations sur points de courbe elliptique sur corps binaire, en comparaison des opérations sur courbe sur corps premier $\mathbb{F}_p$. En particulier, l'efficacité de l'élévation au carré, de la résolution d'équations quadratiques et du calcul des racines carrées ouvre la possibilité de diviser par deux un point de la courbe elliptique dans de bonnes conditions.

Dans cette section, nous présentons donc l'addition et le doublement de points ainsi que deux systèmes de coordonnées de points, puis la division par deux de point. En fin de section, nous fournissons les complexités de toutes ces opérations.

Nous utilisons la forme courte de l'équation de Weierstrass représentant la courbe elliptique sur le corps $\mathbb{F}_{2^m}$. E sur $\mathbb{F}_{2^m}$ munie d'une loi d'addition est un groupe fini tel que :

$$E(\mathbb{F}_{2^m}) = \{(x, y) \in \mathbb{F}_{2^m} \times \mathbb{F}_{2^m} \mid y^2 + xy = x^3 + ax^2 + b\} \cup \{\mathcal{O}\}.$$

avec $(a, b) \in \mathbb{F}_{2^m} \times \mathbb{F}_{2^m}, b \neq 0$.

2.3.3.1 Addition et doublement

Ces deux opérations s'effectuent entre points d'une courbe elliptique, et le résultat est un autre point de la courbe elliptique. Pour plus de détails, le lecteur pourra se reporter aux livre et articles suivants, dont nous tirons les formules ci-après : [28], [22], [24], [33]. L'addition entre deux points d'une courbe elliptique P_1 et P_2 s'écrit :

$$P_3 = P_1 + P_2 \text{ avec } \begin{cases} P_1 = (x_1, y_1), \\ P_2 = (x_2, y_2), \\ P_3 = (x_3, y_3), \end{cases} \quad \text{où : } \quad \begin{cases} x_3 = \lambda^2 + \lambda + x_1 + x_2 + a, \\ y_3 = (x_1 + x_3)\lambda + x_3 + y_1, \end{cases}$$

et :

$$\begin{cases} \lambda = \frac{y_1+y_2}{x_1+x_2} \text{ si } P_1 \neq P_2, \\ \\ \lambda = \frac{y_1}{x_1} + x_1 \text{ si } P_1 = P_2. \end{cases}$$

On note aussi pour finir que le point $-P$, tel que $P + (-P) = \mathcal{O}$, a pour coordonnées $(x, x + y)$

Comme exposé en section 2.3.2.2 page 58 concernant les coordonnées jacobiennes pour les points de courbe elliptique sur $\mathbb{F}_p$, on peut tirer parti de systèmes de coordonnées projectives. La démarche est la même que pour les coordonnées jacobiennes, et les avantages de ces systèmes sont les mêmes, notamment la suppression de l'inversion modulaire, opération coûteuse. Nous présentons maintenant deux systèmes : celui proposé par Lopez-Dahab (voir [22]) et une variante de ce système présentée par Kim et Kim dans [33].

2.3.3.1.1 Coordonnées projectives Lopez-Dahab $\mathcal{LD}$

Ces coordonnées projectives se présentent de la façon suivante :

$$P = (x, y) \text{ en } (X : Y : Z), \text{ avec } \begin{cases} x = \frac{X}{Z}, \\ \\ y = \frac{Y}{Z^2}. \end{cases}$$

Avec ces coordonnées, l'équation de la courbe elliptique devient :

$$Y^2 + XYZ = X^3Z + aX^2Z^2 + bZ^4.$$

Sans rappeler toute la démarche, nous donnons simplement les formules fournies dans [22] : les formules pour le doublement en coordonnées projectives Lopez-Dahab $\mathcal{LD}$ sont données dans la table 2.8, et pour l'addition en coordonnées mixées ($\mathcal{LD}$ et affines, avec $Z_2 = 1$ dans ce cas) dans la table 2.9.

$$2 \cdot (X : Y : Z) = (X_1 : Y_1 : Z_1), \text{ avec } \begin{cases} X_1 = X^4 + b \cdot Z^4, \\ Y_1 = bZ^4 \cdot Z_1 + X_1 \cdot (aZ_1 + Y^2 + bZ^4), \\ Z_1 = X^2 \cdot Z^2. \end{cases}$$

TABLE 2.8 – Formules pour le doublement en coordonnées $\mathcal{LD}$, courbe $E(\mathbb{F}_{2^m})$, d'après les auteurs dans [22].

$$(X_1 : Y_1 : Z_1) + (X_2 : Y_2 : 1) = (X_3 : Y_3 : Z_3),$$

$$\text{avec } \begin{cases} X_3 = A^2 + D + E, \\ Y_3 = E \cdot F + C^2 \cdot G = E \cdot F + Z_3 \cdot G, \\ Z_3 = C^2, \end{cases} \quad \text{où } \begin{cases} A = Y_2 \cdot Z_1^2 + Y_1, \\ B = X_2 \cdot Z_1 + X_1, \\ C = Z_1 \cdot B, \\ D = B^2 \cdot (C + aZ_1^2), \\ E = A \cdot C, \\ F = X_3 + X_2 \cdot Z_3, \\ G = X_3 + Y_2 \cdot Z_3. \end{cases}$$

TABLE 2.9 – Formules pour l'addition en coordonnées mixées ($\mathcal{LD}$ et affines), courbe $E(\mathbb{F}_{2^m})$ d'après les auteurs dans [22].

2.3.3.1.2 Les coordonnées *PL*

Ce système de coordonnées est proposé par Kim et Kim dans [33]. Il s'agit en réalité d'une variante optimisée du précédent. On transforme un point de courbe en coordonnées affines

$$P = (x, y) \text{ en } (X : Y : Z : T), \text{ avec } \begin{cases} x = \frac{X}{Z}, \\ \\ y = \frac{[Y]}{T}, \end{cases} \quad \text{et} \quad T = Z^2.$$

La notation $[Y]$ signifie que l'on effectue une réduction modulaire avec le polynôme $f(x)$ définissant $\mathbb{F}_{2^m}$. Par exemple, pour $m = 233$, ce polynôme est $f(x) = x^{233} + x^{74} + 1$. Y n'est donc

pas réduit, et peut avoir un degré au plus $2 \times 232 = 464$. Avec ces coordonnées, l'équation de la courbe elliptique devient alors :

$$[Y]^2 + X[Y]Z = X^3Z + aX^2Z^2 + bZ^4.$$

En notant $\times$ la multiplication de polynômes sans réduction, nous pouvons écrire le doublement et l'addition comme précédemment. Les formules pour le doublement en coordonnées *PL* figurent dans la table 2.10, et pour l'addition en coordonnées mixées (*PL* et affine) dans la table 2.11.

$$2 \cdot (X : Y : Z : T) = (X_1 : Y_1 : Z_1 : T_1), \text{ avec :}$$

$$\begin{cases} X_1 = [A \times A + b \cdot T^2], \\ Y_1 = B \times (B + X_1 + Z_1) + b \times T_1 + T_1, \\ Z_1 = T \cdot A, \\ T_1 = Z_1^2, \end{cases} \quad \text{où} \quad \begin{cases} A = X^2, \\ B = [Y]^2. \end{cases}$$

TABLE 2.10 – Formules pour le doublement en coordonnées *PL*, courbe $E(\mathbb{F}_{2^m})$ d'après [33].

$$(X_1 : Y_1 : Z_1 : T_1) + (X_2 : Y_2 : 1 : 1) = (X_3 : Y_3 : Z_3 : T_3),$$

$$\text{avec} \begin{cases} X_3 = [B \times B + C \times A^2 + D], \\ Y_3 = (X_3 + X_2 \cdot Z_3) \times D + (X_2 + Y_2) \times T_3, \\ Z_3 = C^2, \\ T_3 = Z_3^2, \end{cases} \quad \text{où} \quad \begin{cases} A = X_1 + X_2 \cdot Z_1, \\ B = [Y_1 + Y_2 \times T_1], \\ C = A \cdot Z_1, \\ D = C \cdot (B + C). \end{cases}$$

TABLE 2.11 – Formules pour l'addition en coordonnées mixées (*PL* et affines), courbe $E(\mathbb{F}_{2^m})$ d'après [33].

2.3.3.2 Division par deux de point, ou *point halving*

On rappelle ici le principe de la division par deux d'un point de courbe elliptique, dont le principe a été proposé indépendamment par Knudsen dans [35] et Schroeppel [57]. Nous nous appuyons sur Fong *et al.* dans [18] et sur Knudsen dans [35] pour exposer l'algorithme de *point halving*, c'est à dire la division par deux d'un point.

En guise de préliminaire, Knudsen dans [35] remarque que l'application de doublement des points n'est pas injective (un élément peut avoir plus d'un antécédent unique) si le groupe sur lequel on l'utilise est d'ordre pair. En effet, dans ce cas, il y a deux points qui ont pour double $\mathcal{O} : \mathcal{O}$ et le point que Knudsen note T_2. Ces deux points forment le noyau de l'application de multiplication par deux, ce qui conduit à se limiter aux points de la courbe qui engendrent un sous-groupe G d'ordre impair. Dans G, l'application de multiplication par deux

est une bijection. Par conséquent, sa réciproque, que Knudsen baptise la *halving map*, est aussi une bijection.

Knudsen dans [35] et Fong *et al.* dans [18] exposent une méthode pour la division par deux d'un point que nous reproduisons ci-après.

Fong *et al.* dans [18] rappellent en premier lieu le doublement $Q = 2 \cdot P = (u, v)$ en coordonnées affines d'un point $P = (x, y) \in E(\mathbb{F}_{2^m})$ tel que $P \neq -P$ est donné par

$$\lambda = x + y/x \tag{2.8}$$
$$u = \lambda^2 + \lambda + a \tag{2.9}$$
$$v = x^2 + u(\lambda + 1) \tag{2.10}$$

L'opération de *halving* est l'opération inverse : connaissant $Q = (u, v)$, on cherche $P = (x, y)$ tel que $Q = 2 \cdot P$. Il faut donc d'abord résoudre l'équation (2.9) pour obtenir λ (en fait, résoudre $\lambda^2+\lambda = u+a$), puis l'équation (2.10) pour obtenir x (soit $x = \sqrt{v + u(\lambda + 1)}$), et enfin l'équation (2.8) pour y (calculer $y = \lambda x + x^2$).

Il reste cependant à déterminer la bonne solution de ces équations de façon à trouver le point élément du sous-groupe G. Pour calculer efficacement cette solution, la fonction *Trace* joue un rôle important :

$$\mathrm{Tr}(c) = c + c^2 + c^{2^2} + \ldots + c^{2^{m-1}}.$$

— $\mathrm{Tr}(c)$ est une fonction linéaire ;
— $\mathrm{Tr}(c) \in \{0, 1\}$;

Lemme 2.3.1. *Soit G le sous-groupe engendré par un élément d'ordre impair de la courbe $E(\mathbb{F}_{2^m})$. Si $(u, v) \in G$, alors $Tr(u) = Tr(a)$.*

Démonstration. Cette relation $\mathrm{Tr}(u) = \mathrm{Tr}(a)$ s'obtient en appliquant la fonction *Trace* à l'équation (2.9) et en remarquant que $\mathrm{Tr}(\lambda^2) = \mathrm{Tr}(\lambda)$, d'où $\mathrm{Tr}(\lambda^2)+\mathrm{Tr}(\lambda) = 0$. □

En particulier, Fong *et al.* donnent le théorème suivant :

Théorème 2.3.1. *Soit G le sous-groupe engendré par un élément d'ordre r impair de la courbe $E(\mathbb{F}_{2^m})$ et $Tr(a) = 1$. Soit $P = (x, y) \in G$ et $Q = (u, v) \in G$ tels que $Q = 2P$ et soit $\lambda = x + y/x$. Soit $\hat{\lambda}$ une solution de l'équation $\lambda^2 + \lambda = u + a$ et $t = v + u\hat{\lambda}$. Alors $\hat{\lambda} = \lambda$ si et seulement si $Tr(t) = 0$.*

Démonstration. On remarque d'abord d'après le lemme 2.3.1 que comme $P, Q \in G$ on a $\mathrm{Tr}(x) = \mathrm{Tr}(u) = \mathrm{Tr}(a) = 1$. On remarque aussi que les deux solutions de l'équation $\lambda^2 + \lambda = u + a$ sont λ et $\lambda + 1$. On a alors deux cas :

— Si $\hat{\lambda} = \lambda$ alors

$$\begin{aligned} \mathrm{Tr}(t) &= \mathrm{Tr}(v + u\lambda) \\ &= \mathrm{Tr}(v + u(\lambda + 1) + u) \\ &= \mathrm{Tr}(x^2 + u) \qquad \text{(d'après (2.10))} \\ &= \mathrm{Tr}(x) + \mathrm{Tr}(u) = 2\mathrm{Tr}(a) = 0. \end{aligned}$$

— Si $\hat{\lambda} = \lambda + 1$ alors d'après (2.10) on a

$$\mathrm{Tr}(t) = \mathrm{Tr}(v + u(\lambda + 1)) = \mathrm{Tr}(x^2) = \mathrm{Tr}(x) = \mathrm{Tr}(a) = 1.$$

□

Ce théorème donne une méthode économique pour déterminer la bonne solution, par le calcul de $\mathrm{Tr}(t)$.

Remarque 2.3.4. *Pour les courbes elliptiques recommandées par le NIST pour usage cryptographique, on a toujours pour* $E(\mathbb{F}_{2^m}) = \{(x, y) \in \mathbb{F}_{2^m} \times \mathbb{F}_{2^m} \mid y^2 + xy = x^3 + ax^2 + b\} \cup \{\mathcal{O}\}$ *(forme courte de Weierstrass) :*

$$Tr(a) = 1.$$

Calcul efficace de la fonction Trace

Soit $c = \sum_{i=0}^{m-1} c_i z^i \in \mathbb{F}_{2^m}$. On peut calculer $\mathrm{Tr}(c)$ en utilisant la linéarité de la fonction Trace. On peut écrire $\mathrm{Tr}(c) = \sum_{i=0}^{m-1} c_i \mathrm{Tr}(x^i)$. Les valeurs de $\mathrm{Tr}(x^i)$ peuvent être calculées préalablement, et le calcul est d'autant plus rapide si une grande partie de ces valeurs est nulle. C'est ce que montre l'exemple suivant.

Exemple 2.3.1. *Pour définir le corps* $\mathbb{F}_{2^{233}}$*, le NIST dans [19] recommande le polynôme de réduction* $f(x) = x^{233} + x^{74} + 1$*. On peut vérifier que* $Tr(x^i) = 1$ *si et seulement si* $i \in \{0, 159\}$*.*

Il reste ensuite la résolution proprement dite de notre système formé par les équations (2.8), (2.9) et (2.10).

Solution de l'équation quadratique (2.9)

Pour la résolution de l'équation quadratique (2.9), Fong *et al.* dans [18] font usage de la fonction *half-trace*. Ils démontrent le lemme suivant :

Lemme 2.3.2. *On suppose que* m *est impair. On définit la fonction* half-trace $H : \mathbb{F}_{2^m} \to \mathbb{F}_{2^m}$ *telle que*

$$H(c) = \sum_{i=0}^{(m-1)/2} c^{2^{2i}}.$$

On a :

(i) $H(c)$ *est linéaire ;*

(ii) $H(c)$ *est une solution de* $x^2 + x = c + Tr(c)$ *;*

(iii) $H(c) = H(c^2) + c + Tr(c)$ *pour tout* $c \in \mathbb{F}_{2^m}$*.*

Le calcul de la fonction *half-trace* est efficace en utilisant, comme pour la fonction *Trace*, sa propriété de linéarité, et des tables calculées au préalable.

Algorithme de division par deux de point et complexité

Pour le *point halving*, nous donnons l'algorithme 2.3.

En terme de complexité, on admet les résultats suivants :

— La résolution de l'équation (2.9), en particulier le calcul de la fonction *half-trace*, coûte environ 1,5 multiplications d'éléments du corps $\mathbb{F}_{2^m}$;

— Le calcul d'une racine carrée coûte environ 0,5 multiplication d'éléments du corps $\mathbb{F}_{2^m}$.

La complexité totale du *halving* est de deux multiplications avec réduction, un calcul de racine carrée et une résolution d'équation quadratique. Avec les hypothèses ci-dessus, cela fait donc au total l'équivalent d'environ quatre multiplications avec réductions.

Pour économiser une multiplication, Fong *et al.* proposent de représenter le point $Q = (u, v)$ par les coordonnées (u, λ_Q) avec $\lambda_Q = u + v/u$, qu'ils baptisent $\lambda - representation$. La complexité devient alors la suivante : une multiplication avec réduction, un calcul de racine carrée et une résolution d'équation quadratique. Avec les hypothèses ci-dessus, cela fait donc au total l'équivalent d'environ trois multiplications avec réductions.

Algorithme 2.3 *Halving* de point de courbe elliptique sur $\mathbb{F}_{2^m}$ d'ordre impair

Require: $Q = (u, v)$ en coordonnées affines, $Q \in G$.
Ensure: $P = (x, y) \in G$, où $Q = 2 \cdot P$
1: Calculer une solution de $\hat{\lambda}$ of $\hat{\lambda}^2 + \hat{\lambda} = u + a$
2: $t \leftarrow v + u\hat{\lambda}$
3: **if** $\mathrm{Tr}(t) = 0$ **then**
4: $\quad \lambda_P \leftarrow \hat{\lambda}, x \leftarrow \sqrt{t + u}$
5: **else**
6: $\quad \lambda_P \leftarrow \hat{\lambda} + 1, x \leftarrow \sqrt{t}$
7: **end if**
8: $y \leftarrow x\lambda_P + x$
9: **return** (x, y)

2.3.3.3 Bilan de complexité

Ce qu'il faut retenir, c'est que le *halving* est plus efficace que le doublement. Dans le cas de la caractéristique 2, l'utilisation de cette opération dans une multiplication scalaire présente un intérêt certain.

Nous présentons ici dans la table 2.12 un petit bilan de complexité des opérations sur la courbe. L'opération la plus économique est bien le *Point Halving*, suivi du doublement, puis de l'addition en coordonnées mixées dans la variante PL. Enfin, les opérations en coordonnées affines ferment la marche, en raison du coût de l'inversion.

Operation	Système de Coord.	Coût
Doublement	Affine	$1I + 2M + 1S + 3R$
Addition	Affine	$1I + 2M + 1S + 3R$
Doublement	$\mathcal{LD}$	$4M + 4S + 8R$
Addition en coordonnées mixées	$\mathcal{LD}$	$9M + 4S + 13R$
Addition Projective	$\mathcal{LD}$	$13M + 4S + 17R$
Doublement	PL	$4M + 5S + 7R$
Addition en coordonnées mixées	PL	$8M + 4S + 9R$
halving	Affine	$2M + 1SR + 2R + 1QS$
halving	$\lambda - representation$	$1M + 1SR + 1R + 1QS$

I = inversion, M = multiplications, S = élévations au carré, SR = racines carrées,
QS = résolution de l'équation quadratique, R = réduction

TABLE 2.12 – Complexité des opérations de points de courbe elliptique sur $\mathbb{F}_{2^m}$.

2.3.4 Multiplication scalaire

Après cette revue des opérations sur points de courbe elliptique sur $\mathbb{F}_p$ et $\mathbb{F}_{2^m}$, nous pouvons nous attaquer à la question qui nous motive : la multiplication scalaire. Nous entrons dans le détail des algorithmes les plus classiques basés sur le *Double-and-add* présentés dans la

littérature, avec notamment, les optimisations *Non Adjacent Form* (NAF) et sa variante γ-NAF. Nous présentons aussi, dans le cas particulier des courbes elliptiques sur $\mathbb{F}_{2^m}$, l'algorithme *Halve-and-add* et la version parallèle correspondante.

2.3.4.1 Algorithme de base : *Double-and-add*

Pour effectuer un produit scalaire $k \cdot P$, nous pouvons additionner $k-1$ fois P à lui-même. Nous avons vu que k est grand, et il semble donc peu recommandé de procéder ainsi.

Le procédé courant exposé dans la littérature, notamment par Hankerson *et al.* dans [22] est l'algorithme *Double-and-add* dans ses deux variantes *Left-to-right* et *Right-to-left*, selon le sens adopté pour parcourir les bits de la représentation du scalaire (algorithme 2.4 et 2.5). Le principe est analogue à celui de l'exponentiation rapide (*Square-and-multiply*), utilisé pour le protocole *RSA* par exemple. En effet, la comparaison avec l'algorithme 2.1 ou 2.2 page 53 fait apparaître que si l'on remplace les doublements et additions de points respectivement par des carrés et multiplications modulaires d'éléments de $\mathbb{Z}/N\mathbb{Z}$, les deux algorithmes sont similaires.

Algorithme 2.4 *Left-to-right Double-and-add*

Require: $k = (k_{t-1}, ..., k_1, k_0), P \in E(\mathbb{F}_q)$
Ensure: $Q = k \cdot P$
1: $Q \leftarrow \mathcal{O}$
2: **for** $i = t-1$ downto 0 **do**
3: $\quad Q \leftarrow 2 \cdot Q$
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad Q \leftarrow Q + P$
6: $\quad$ **end if**
7: **end for**
8: **return** (Q)

Algorithme 2.5 *Right-to-left Double-and-add*

Require: $k = (k_{t-1}, ..., k_1, k_0), P \in E(\mathbb{F}_q)$
Ensure: $Q = k \cdot P$
1: $Q \leftarrow \mathcal{O}$
2: **for** $i = 0$ to $t-1$ **do**
3: $\quad$ **if** $k_i = 1$ **then**
4: $\quad\quad Q \leftarrow Q + P$
5: $\quad$ **end if**
6: $\quad P \leftarrow 2 \cdot P$
7: **end for**
8: **return** (Q)

La complexité de cet algorithme, pour un scalaire de longueur t est de t doublements et en moyenne de $t/2$ additions de points, pour un scalaire tiré au sort.

On note ici que les deux variantes ne sont pas équivalentes, contrairement à ce que l'on avait vu dans le cas de l'exponentiation modulaire. En effet, l'addition de point étape 5 dans le cas *Left-to-right* est une addition en coordonnées mixées, tandis que dans le cas *Right-to-left*, l'addition étape 4 est en coordonnées projectives pour les deux points (jacobiennes pour les courbe sur corps de grande caractéristique, coordonnées *PL* ou $\mathcal{LD}$ pour les courbes sur corps binaires). La version *Right-to-left* est donc plus coûteuse dans ce cas.

2.3.4.2 NAF : *Non Adjacent Form*

Cette forme de codage d'un entier vient de la remarque suivante : $\forall k \in \mathbb{N}$ tel que $k = 2^i - 1$, on a l'expression en binaire :

$$(k)_2 = \underbrace{111...1}_{i \text{ fois}}.$$

On peut alors écrire :

$$(k)_{NAF} = \underbrace{100...00 - 1}_{i+1 \text{ chiffres}}. \tag{2.11}$$

La représentation NAF est donc une transformation telle que : pour $k \in \mathbb{N},\ k = (k_{t-1}, ..., k_1, k_0)_2$ avec $k_i \in \{0, 1\}$,

$$k = \sum_{i=0}^{t} k'_i 2^i \text{ où } k'_i \in \{0, +1, -1\}$$

Cette représentation utilise le chiffre -1 en plus, en revanche, elle diminue le nombre de chiffres non nuls dans la représentation d'un entier. Elle élimine en effet tous les 1 consécutifs, et les remplace comme nous venons de le voir dans l'équation (2.11). Ceci permet de réduire d'autant le nombre d'additions de points de courbe elliptique, et donc le temps de calcul. La proportion moyenne de chiffres non nuls passe de $1/2$ à $1/3$. Hankerson *et al.* dans [22] décrivent un algorithme permettant de calculer cette représentation. C'est l'algorithme 2.6.

Algorithme 2.6 Calcul de la répresentation NAF d'un entier positif

Require: $k \in \mathbb{N}$
Ensure: NAF(k)
1: $i \leftarrow 0$
2: **while** $k \geq 1$ **do**
3: **if** k impair **then**
4: $k_i \leftarrow 2 - (k \bmod 4), k \leftarrow k - k_i$
5: **else**
6: $k_i \leftarrow 0$
7: **end if**
8: $k \leftarrow k/2, i \leftarrow i + 1$
9: **end while**
10: **return** $(k_{i-1}, k_{i-2}, ..., k_1, k_0)$

Le *Double-and-add* est alors modifié comme montré dans l'algorithme 2.7, toujours d'après Hankerson *et al.* dans [22], afin de prendre en compte les coefficients valant -1 de la représentation du scalaire. La complexité de cet algorithme, pour un scalaire de longueur t est de t doublements et en moyenne de $t/3$ additions de points, pour un scalaire tiré au sort.

Remarque 2.3.5. *L'algorithme 2.7 requiert le calcul de la soustraction $Q \leftarrow Q - P$. Cette opération est effectuée comme suit :*

$$Q \leftarrow Q - P = Q + (-P),$$

en remarquant que :
— *dans le cas d'une courbe $E(\mathbb{F}_p)$, pour $P = (x, y) \in E$, on a : $-P = (x, -y)$.*
— *dans le cas d'une courbe $E(\mathbb{F}_{2^m})$, pour $P = (x, y) \in E$, on a : $-P = (x, x + y)$.*

Algorithme 2.7 NAF *Left-to-right Double-and-add*

Require: $k = (k_{t-1}, ..., k_1, k_0)$ représentation *NAF* de k, $P \in E(\mathbb{F}_{2^m})$.
Ensure: $Q = k \cdot P$
1: $Q \leftarrow \mathcal{O}$
2: **for** $i = t - 1$ downto 0 **do**
3: $Q \leftarrow 2 \cdot Q$
4: **if** $k_i = 1$ **then**
5: $Q \leftarrow Q + P$
6: **end if**
7: **if** $k_i = -1$ **then**
8: $Q \leftarrow Q - P$
9: **end if**
10: **end for**
11: **return** (Q)

2.3.4.3 γ-NAF

Cette représentation est une extension de la représentation NAF. En effet, plutôt que de limiter le codage de chaque chiffre à l'ensemble $\{-1, 0, 1\}$, on utilise l'ensemble :

$$\{-2^{\gamma-1} + 1, ..., -5, -3, -1, 0, 1, 3, 5, ..., 2^{\gamma-1} - 1\}.$$

Cette astuce augmente significativement le nombre de chiffres nuls. Il n'en reste qu'une proportion de $1/(\gamma + 1)$ en moyenne. L'algorithme de cette fonction se base sur le même principe que l'algorithme 2.6 (page 69). Le *Double-and-add* est quant à lui modifié comme montré dans l'algorithme 2.8 (toujours d'après Hankerson *et al.* dans [22]) :

Algorithme 2.8 γ-NAF *Left-to-right Double-and-add*

Require: Fenêtre de largeur γ, γ-NAF$(k) = \sum_{i=0}^{t-1} k_i \cdot 2^i$, $P \in E(\mathbb{F}_{2^m})$.
Ensure: $Q = k \cdot P$
1: Calculs de $P_i = i \cdot P$ pour $i \in \{1, 3, 5, ..., 2^{\gamma-1} - 1\}$
2: $Q \leftarrow \mathcal{O}$
3: **for** $i = t - 1$ downto 0 **do**
4: $Q \leftarrow 2 \cdot Q$
5: **if** $k_i \neq 0$ **then**
6: **if** $k_i > 0$ **then**
7: $Q \leftarrow Q + P_{k_i}$
8: **else**
9: $Q \leftarrow Q - P_{k_i}$
10: **end if**
11: **end if**
12: **end for**
13: **return** (Q)

Par rapport à la représentation NAF, il faut calculer en plus la table des

$$\{(-2^{\gamma-1} + 1) \cdot P, ..., -5P, -3P, -P, 0, P, 3P, 5P, ..., (2^{\gamma-1} - 1) \cdot P\}$$

où P est le point générateur de notre courbe elliptique. Ceci nécessite $2^{\gamma-2}$ additions de points. La complexité de l'algorithme 2.8, pour un scalaire de longueur t est de t doublements et en moyenne de $t/(\gamma + 1) + 2^{\gamma-2}$ additions de points, pour un scalaire tiré au sort.

2.3.4.4 L'algorithme *Halve-and-add* et parallèle *Double/halve-and-add* dans $\mathbb{F}_{2^m}$

Nous avons vu section 2.3.3.2 page 64 que dans le cas des courbes elliptiques sur $\mathbb{F}_{2^m}$, l'opération de division par deux de points (*halving*) était plus efficace que le doublement. On présente ici l'algorithme de multiplication scalaire de points de courbe elliptique utilisant le *halving* au lieu du doublement.

Dans cette approche, la multiplication scalaire va être constituée d'une séquence de *halving* et d'additions de points de la courbe. Soit r l'ordre impair des points de la courbe ($r = \#< P >$). Le scalaire k de longueur t est récrit en $k' = k \cdot 2^{t-1} \bmod r$. En effet, nous avons dans ce cas :

$$k = k'2^{-(t-1)} = \left(\sum_{i=0}^{t-1} k'_i 2^i\right) 2^{-(t-1)} = \sum_{i=0}^{t-1} k'_i 2^{i-(t-1)}$$

et $k \cdot P$ peut ainsi être calculé avec des *halvings*, ayant remplacé les puissances positives de 2 par des puissances négatives de 2, ou encore des puissances positives de $\frac{1}{2}$. Ceci donne l'algorithme 2.9, baptisé *Halve-and-add*.

Algorithme 2.9 *Halve-and-add*

Require: $k = (k_{t-1}, ..., k_1, k_0)$ représentation binaire de k, $P \in E(\mathbb{F}_{2^m})$.
Ensure: $Q = k \cdot P$
1: $Q \leftarrow \mathcal{O}$
2: Réécriture : $k' = k \cdot 2^{t-1} \bmod r$
3: **for** $i = t - 1$ downto 0 **do**
4: **if** $k_i = 1$ **then**
5: $Q \leftarrow Q + P$
6: **end if**
7: $P \leftarrow P/2$
8: **end for**
9: **return** (Q)

Comme dans le cas précédent (*Double-and-add*, les algorithmes 2.7 et 2.8), on peut extrapoler des variantes de l'algorithme 2.9 exploitant la représentation NAF et γ-NAF.

Taverne *et al.* introduisent dans [65] une intéressante version parallèle de la multiplication scalaire. La parallélisation qu'ils proposent consiste à séparer la multiplication scalaire en deux *threads* indépendants. Pour ce faire, ils récrivent le scalaire de longueur t-bits en $k = k_1 + k_2$ où k_1 et k_2 sont calculés comme suit

$$k = \underbrace{(k'_t 2^{t-\ell} + \ldots + k'_\ell)}_{k_1} + \underbrace{(k'_{\ell-1} 2^{-1} + \ldots + k'_0 2^{-\ell})}_{k_2}. \tag{2.12}$$

k_1 est représenté par des puissances positives de 2, k_2 par des puissances négatives de 2 (ou positives de $\frac{1}{2}$).

En général ℓ est proche de $t/2$. Ainsi, la multiplication peut s'effectuer à l'aide d'un calcul partiel $k_1 P$ avec l'algorithme *Double-and-add* et un deuxième calcul partiel $k_2 P$ avec l'algorithme *Halve-and-add*.

Cette méthode est décrite dans l'algorithme 2.10 présenté ici pour un scalaire en représentation γ-NAF.

On remarque que cet algorithme nécessite une étape de reconstruction finale (étape 25). Sa complexité globale est donc légèrement supérieure à celle de l'approche *Halve-and-add* par exemple (algorithme 2.9). Nous verrons cependant que les publications qui font état de son implantation en font l'algorithme le plus rapide de l'état de l'art pour les courbes elliptiques de Weierstrass sur corps binaire.

Algorithme 2.10 *Double/halve-and-add*, multiplication scalaire parallélisée

Require: scalaire k, $P \in \mathbb{F}_{2^m}$ d'ordre r impair, n constant.
Ensure: kP

1: Réécriture : $k' = 2^\ell k \bmod r$, en représentation γ-NAF $k = \sum_{i=0}^{t} k'_i 2^{i-\ell}$
2: Calcul préalable de $P_i = i \cdot P$ pour $i \in I = \{1, 3, 5, ..., 2^{\gamma-1} - 1\}$
3: $Q_D \leftarrow \mathcal{O}$
4: $Q_{Hi} \leftarrow \mathcal{O}$ for $i \in I$ (Barrier)

5: **for** $i = t$ downto ℓ **do**
6: $\quad Q_D \leftarrow 2 \cdot Q_D$
7: $\quad$ **if** $k'_i \neq 0$ **then**
8: $\quad\quad$ **if** $k'_i > 0$ **then**
9: $\quad\quad\quad Q_D \leftarrow Q_D + P_{k'_i}$
10: $\quad\quad$ **else**
11: $\quad\quad\quad Q_D \leftarrow Q_D - P_{k'_i}$
12: $\quad\quad$ **end if**
13: $\quad$ **end if**
14: **end for**

(Barrier)
25: **return** $Q \leftarrow Q_D + \sum_{i \in I} i Q_{Hi}$

15: **for** $i = \ell - 1$ downto 0 **do**
16: $\quad P \leftarrow P/2$
17: $\quad$ **if** $k'_i \neq 0$ **then**
18: $\quad\quad$ **if** $k'_i > 0$ **then**
19: $\quad\quad\quad Q_{Hk'_i} \leftarrow Q_{Hk'_i} + P$
20: $\quad\quad$ **else**
21: $\quad\quad\quad Q_{Hk'_i} \leftarrow Q_{Hk'_i} - P$
22: $\quad\quad$ **end if**
23: $\quad$ **end if**
24: **end for**

2.3.5 Bilan final

Le bilan de complexité final pour l'ensemble des algorithmes de multiplication scalaire de point de courbe elliptique est donné dans la table 2.13.

Il est utile de rappeler que les opérations sur les points de courbe (addition, doublement ou *halving*) ont elles-mêmes leur propre complexité en matière d'opérations sur le corps $\mathbb{F}_p$ ou $\mathbb{F}_{2^m}$. Nous renvoyons le lecteur aux complexités que nous avons données dans les tables 2.7 page 61 et 2.12 page 67.

	nb. de doublements	nb. de *halvings*	nb. d'additions	Courbe
Double-and-add	t	-	$t/2$	$E(\mathbb{F}_p), E(\mathbb{F}_{2^m})$
NAF Double-and-add	t	-	$t/3$	
γ-*NAF Double-and-add*	t	-	$t/(\gamma+1)+2^{\gamma-2}$	
Halve-and-add	-	t	$t/2$	$E(\mathbb{F}_{2^m})$
NAF Halve-and-add	-	t	$t/3$	
γ-*NAF Halve-and-add*	-	t	$t/(\gamma+1)+2^{\gamma-2}$	
γ-*NAF //Double-Halve-and-add* split ℓ	$t-\ell$	ℓ	$t/(\gamma+1)+2^{\gamma-1}$	

TABLE 2.13 – Bilan de complexité pour la multiplication scalaire de points de courbe elliptique.

Chapitre 3

Revue des attaques par canal auxiliaire et contre-mesures

Pour faire suite au précédent chapitre portant principalement sur les aspects arithmétiques liés aux protocoles, le présent chapitre se consacre à la sécurité de ces protocoles, en particulier à leur cryptanalyse en relation avec ce que l'on appelle les attaques par canal auxiliaire (en anglais *side-channel*). En effet, nous avions examiné succinctement la complexité de deux problèmes difficiles qu'un attaquant doit résoudre pour casser les différents protocoles vus au chapitre précédent, qui sont le problème de la factorisation et celui du logarithme discret. Le canal auxiliaire est une source d'information qui permet de contourner tout ou partie de la complexité de ces problèmes. Et en conséquence, nous serons amenés à considérer les questions d'implantations, de plates-formes et d'environnement physique de ces plates-formes.

Nous donnerons premièrement une vue générale de ces attaques avec leur classification, puis en examinerons principalement une, celle baptisée *Simple Power Analysis* ou SPA, sachant que l'essentiel des contributions de notre travail de thèse concerne des problématiques liées à cette attaque. Nous dirons quelques mots d'autres attaques plus sophistiquées, en particulier nous introduirons la *Timing Attack* puis l'attaque *Differential Power Analysis* ou DPA.

3.1 Attaques par canal auxiliaire, vue générale

Un canal auxiliaire, de quoi s'agit-il ? Un exemple tiré des films de série B, c'est celui du cambrioleur qui ouvre un coffre-fort en tournant les tambours de sélection de la combinaison. S'il doit essayer toutes les combinaisons possibles, il lui est impossible de parvenir à ouvrir le coffre, à moins d'y passer des heures voire des jours... Mais s'il se munit d'un stéthoscope, il peut entendre les imperceptibles cliquetis mécaniques qui trahissent le passage d'un disque devant la lumière correspondant au chiffre de la combinaison. Et en quelques minutes à peine (dans les films, c'est même plus rapide, les spectateurs ne supportent guère un suspense trop long !), voilà notre cambrioleur qui ouvre le coffre ! C'est possible grâce à cette source supplémentaire d'information que constitue le bruit du mécanisme, et que l'on nomme canal auxiliaire.

L'Histoire attribue aux services de renseignements britanniques (voir [34]), le MI-5, la première attaque par canal auxiliaire. En 1956, cette agence de renseignement ne parvenait pas à casser le chiffre utilisé par l'ambassade égyptienne jusqu'au jour où Peter Wright, un de leurs mathématiciens, a proposé de placer un micro près de la machine à chiffrer. Cette dernière était munie de rotors qui déterminaient la clé de chiffrement. Chaque matin, le responsable des communications réglait la machine et le micro transmettait les clics des rotors. Ceci a permis aux services de renseignements britanniques d'espionner les communications de l'ambassade

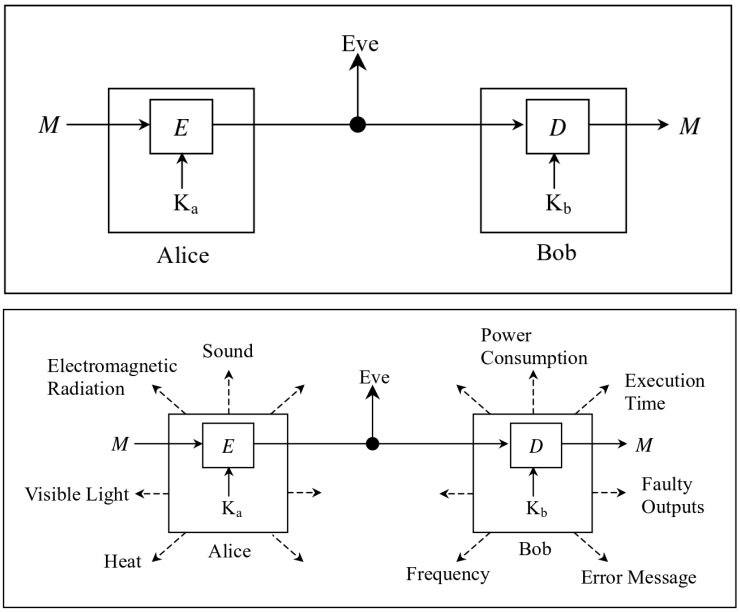


FIGURE 3.1 – Modèle de sécurité traditionnel de la cryptographie (en haut) et modèle incluant les canaux auxiliaires (en bas), tiré de [67].

égyptienne, et ce des années durant ! Autrement, la difficulté calculatoire ne permettait pas de trouver la clé et de déchiffrer les messages en temps raisonnable...

De nos jours, les communications dont nous parlons sont chiffrées par des ordinateurs ou des calculateurs au sens large. Dans le cas des protocoles cryptographiques, le canal auxiliaire va dépendre de la plate-forme ou de l'appareil qui effectue les calculs. D'après la littérature, on peut dresser une liste de ces différents canaux auxiliaires possibles. Celle que nous donnons ci-après ne prétend nullement être exhaustive :

— courant ou puissance électrique instantanés ;
— rayonnement électromagnétique, y compris émissions lumineuses ;
— informations systèmes telles que défauts de cache, prédiction de sauts valides ou non, temps ou nombre de cycles du calcul...
— bruit émis par le processeur (analogue aux claquements de condensateurs) ;
— chaleur rayonnée ;
— etc.

Tous ces phénomènes physiques sont plus ou moins importants selon la plate-forme attaquée. Ainsi, les systèmes embarqués ou mobiles (cartes à puces, téléphones mobiles) vont être plus vulnérables à des mesures de courant ou électromagnétiques, du fait de leur petite taille, de la faible fréquence d'horloge et de la simplicité de leur système d'exploitation (mono-utilisateur et monotâche, parfois). Au contraire, des attaques par le biais du système deviennent plus pertinentes si on s'en prend à de «grosses» machines, par exemple des serveurs. Dans ce dernier cas, les attaques par le comptage de défauts de cache deviennent de vraies menaces, là où le courant instantané consommé devient difficile voire impossible à mesurer. C'est vrai dans le cas d'une machine importante et travaillant à fréquence d'horloge élevée, avec un système complexe multiutilisateurs et multitâches, ce qui génère beaucoup de bruit. Nous reproduisons dans la figure 3.1 les schémas de Zhou *et al.* dans [67] qui illustrent la différence entre le modèle de sécurité traditionnel de la cryptographie et le modèle incluant les canaux auxiliaires.

Si l'on voit bien apparaître le besoin de classification de ces attaques par canal auxiliaire, il est aussi nécessaire de classer les contre-mesures en conséquence. Nous présentons une classification générale de ces attaques et ensuite les attaques passives puis actives.

3.1.1 Classification générale

L'observation du canal auxiliaire nécessite toutefois d'instrumenter l'appareil. Zhou *et al.* dans [67] retiennent les trois catégories suivantes selon la puissance conférée à l'attaquant :

- **attaques invasives :** l'attaquant a un accès direct aux composants. Il peut démonter l'appareil, percer des trous dans le boitier pour laisser un passage à une sonde, il peut instrumenter un bus de données pour lire les informations transférées depuis la mémoire vers le (ou l'un des) processeur(s)... Protéger un appareil contre ce type d'agression peut aller jusqu'à nécessiter des mesures d'auto-destruction !
- **attaques non-invasives :** l'attaquant ne peut observer que des phénomènes externes à l'appareil attaqué. Ces phénomènes fournissent des informations laissées disponibles de façon totalement involontaire. Un exemple de ce type d'attaques, c'est la *Timing-attack*. L'attaquant chronomètre le temps de calcul pour le déchiffrement ou l'échange de clé, et en déduit des informations sur la valeur de la clé ou de l'exposant secret utilisé. Contrairement à ce que l'on pourrait penser *a priori*, ces attaques sont des menaces plus sérieuses que les attaques invasives. En effet, si les informations recueillies semblent moins nombreuses, ces attaques sont à la fois indétectables et peu coûteuses. Premièrement, l'indétectabilité rend parfois les contre-mesures difficiles à mettre en œuvre : comment se protéger d'une agression qu'on ne soupçonne pas ? C'est bien l'exemple de l'attaque du MI-5 contre l'ambassade égyptienne. Elle a été efficace pendant des années, les diplomates égyptiens restant persuadés de la sûreté de leurs communications ! Deuxièmement, le faible coût les rend aussi fatalement plus nombreuses. Zhou *et al.* n'hésitent pas à affirmer que cette menace est l'une des plus importantes vis-à-vis de l'industrie des cartes à puce.
- **attaques semi-invasives :** ce concept se situe entre les deux précédents. L'attaquant accède physiquement à l'appareil, mais ne peut le modifier. Typiquement, il peut réaliser un contact électrique, mais uniquement avec les surfaces autorisées (un connecteur par exemple). Ces attaques ne sont pas entièrement indétectables. En effet, dans ce dernier cas, une perturbation de l'alimentation aussi minime soit-elle peut s'avérer visible.

3.1.2 Attaques passives

Comme leur nom l'indique, la mise en œuvre de ces attaques se fait sans agir sur l'appareil qui calcule, l'attaquant ne fait qu'observer passivement. Ces attaques peuvent être non-invasives ou semi-invasives, car l'attaquant opère le plus souvent sur un appareil en situation d'utilisation normale. Les méthodes d'analyse constituent le principal critère de classification, selon le nombre d'échantillons que va traiter l'attaquant et dont il peut disposer. Zhou *et al.* dans [67] distinguent les *Simple Side-Channel Analysis* (SSCA) et les *Differential Side-Channel Analysis* (DSCA).

- ***Simple Side-Channel Analysis* :** l'attaquant ne dispose que d'une seule trace de mesure (par exemple la puissance instantantanée) et éventuellement, mais pas obligatoirement, la paire correspondante texte clair-texte chiffré. Cette attaque réussit à la condition que les informations recueillies par le canal auxiliaire dépendent de la donnée secrète traitée, ou encore que les opérations effectuées révèlent cette donnée. Dans ce cas, le secret se lit quasiment d'un simple coup d'œil sur la trace enregistrée sur le canal auxiliaire. Zhou *et al.* dans [67] font remarquer par ailleurs que le bruit ou les informations non

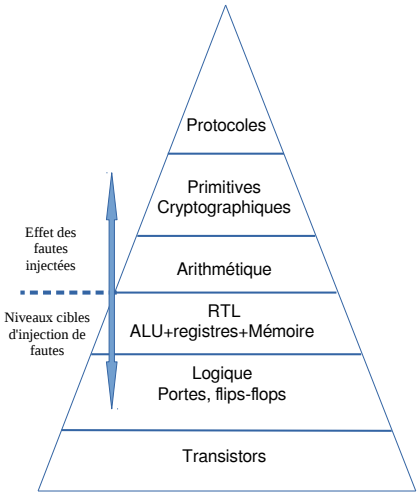


FIGURE 3.2 – La pyramide de la sécurité, reproduite de [66].

pertinentes pour l'attaquant ne doivent pas masquer l'information utile, ou encore que la relation entre les instructions exécutées et le signal issu du canal auxiliaire doit être exploitable.

— ***Differential Side-Channel Analysis*** **:** l'attaquant peut maintenant lancer plusieurs calculs, et obtenir plusieurs traces ainsi que les paires correspondantes textes clairs-textes chiffrés. Ainsi, lorsque la SSCA est inopérante pour des raisons de bruit ou de non corrélation entre instructions exécutées et le signal issu du canal auxiliaire, la possibilité d'effectuer un traitement statistique permet de retrouver la clé secrète probable. Ce traitement peut aussi tirer avantage d'un modèle physique de l'appareil. Le degré de raffinement apporté par ce modèle permet souvent de diminuer le nombre de traces nécessaires pour parvenir à casser le chiffrement. Ce modèle prédit la sortie du canal auxiliaire en fonction d'hypothèses sur la donnée secrète et ainsi, par corrélation avec les traces mesurées, on en déduit la valeur secrète la plus probable.

Dans ce présent travail de thèse, nous avons principalement travaillé sur les améliorations d'algorithmes en relation avec la SSCA.

3.1.3 Attaques actives

Verbauwhede *et al.* dans [66] décrivent les principales attaques et donnent une classification des attaques actives, dites aussi attaques par injection de fautes. Ils proposent un schéma, la pyramide de la sécurité (reproduit figure 3.2) qui permet de visualiser les différents niveaux présents dans les échanges cryptographiques. Verbauwhede *et al.* font une remarque préliminaire : ainsi qu'on peut le voir sur le schéma, les attaques actives interviennent aux plus bas niveaux, depuis les transistors (le niveau le plus bas) jusqu'aux registres, mémoires, unités de calculs, mais la manifestation de ces fautes injectées se propage et se révèle aux niveaux plus hauts (depuis l'arithmétique jusqu'aux protocoles).

Verbauwhede *et al.* classent ensuite les attaques par fautes en fonction de la cible que l'attaquant choisit pour les injecter, et du niveau où le contrôle des conséquences de ces erreurs permet la cryptanalyse. En effet, un processeur traite des données selon un certain contrôle, et

ces trois composantes (données, traitement, contrôle) peuvent faire l'objet d'injection d'erreurs.

Un deuxième critère de classification des attaques mentionné par Verbauwhede *et al.* concerne les différents types d'erreur que l'attaquant peut injecter :

- ***bit flip/set :*** c'est ici que l'attaquant a le plus de puissance ! Il peut changer ou forcer un bit individuel d'une donnée à un moment précis du calcul (changer le signe d'un entier, par exemple).
- **attaque sur un mot :** l'attaquant peut changer ou perturber un mot complet, selon la taille des registres ou des emplacements mémoires. La cible peut être un registre du processeur.
- **variable :** on peut relâcher encore des contraintes. L'attaquant perturbe une variable utilisée en cours de calcul.

Enfin, en troisième critère, l'attaquant peut voir à l'un des deux niveaux les plus hauts (protocole ou primitives cryptographiques) les conséquences de la ou des erreurs qu'il a injectées. On a donc trois critères de classification des attaques.

Verbauwhede *et al.* dans [66] continuent leur revue en proposant une classification similaire pour les contre-mesures. Ainsi, selon le niveau d'abstraction dans lequel on se situe dans la pyramide de la sécurité, on peut appliquer une contre-mesure :

- au niveau du protocole, adapté pour permettre une détection d'erreur avant qu'il se termine ;
- au niveau des primitives cryptographiques, en vérifiant chaque résultat intermédiaire ;
- au niveau arithmétique, en contrôlant chaque opération arithmétique susceptible d'être attaquée.

Si l'on se place aux niveaux inférieurs dans la pyramide de la sécurité, les possibilités sont les suivantes :

- contrôle de l'intégrité des données d'entrées ;
- redondance au niveau des calculs, pour pouvoir détecter les manipulations malicieuses éventuelles ;
- propriétés de l'algorithme dont on peut enfin tirer parti.

3.2 SPA : *Simple Power Analysis*

La première attaque que nous examinons en détail est du type *Simple Side-Channel Analysis*. Cette attaque est décrite par Kocher *et al.* en [39] et en [38]. Nous reprenons cette description. Elle consiste à mesurer l'énergie consommée par un appareil effectuant une opération cryptographique (chiffrement, déchiffrement, signature...) Le principe est d'utiliser ces traces pour en tirer des informations sur les paramètres, notamment une clé secrète.

Cette attaque ne nécessite aucune connaissance préalable d'un texte clair ou chiffré. Quand elle est possible, elle semble donc extrêmement puissante. Avec peu de moyens et quasiment instantanément, on accède à des informations secrètes impossibles à obtenir par des moyens calculatoires (en temps raisonnable). Des appareils qui échantillonnent à 20 MHz ou plus et qui transférent leurs mesures dans un simple PC se trouvent pour moins de 400 US$ d'après les auteurs dans [38], tandis que des instruments de laboratoire mesurent des signaux de l'ordre du GHz avec des précisions de l'ordre de 1%. Ceci est possible pour tous types de plates-formes, depuis les cartes à puces jusqu'aux machines de bureau et autres serveurs d'institutions, en passant pour les FPGA, ASICs...

Nous ne décrivons pas en détail la sœur quasi jumelle de la SPA, la *Simple Electro Magnetic Analysis* (SEMA). Cette attaque ne diffère de la SPA que par le canal auxiliaire. Il ne s'agit pas de la puissance électrique consommée, mais du rayonnement électromagnétique de l'appareil en cours de calcul, comme son nom l'indique. Si la forme du signal et les motifs diffèrent

quelque peu, l'interprétation qu'on en tire est identique. Et les contre-mesures s'appliquent de façon identique à la SPA comme à la SEMA.

Dans la suite de cette section, nous décrivons la SPA, puis présentons les principales contre-mesures.

3.2.1 Description de l'attaque

Reprenons l'exponentiation modulaire *Left-to-right Square-and-multiply*, que nous avons présenté dans l'algorithme 2.1 page 53. Cette opération est utilisée dans le protocole RSA pour le chiffrement et le déchiffrement. Le fonctionnement de cet algorithme fait apparaître une boucle principale au cours de laquelle une élévation au carré est effectuée à chaque itération de la boucle. Un test est ensuite réalisé pour déterminer la nécessité d'effectuer ou non une multiplication. Pour un exposant k tiré aléatoirement et représenté binairement par t bits, effectuer une exponentiation se traduit par t élévations au carré et $t/2$ multiplications en moyenne. Nous avons aussi vu que l'élévation au carré et la multiplication modulaire n'avaient pas la même complexité, et que les algorithmes utilisés étaient différents pour chacune de ces opérations (voir la section 1.1.3).

La question est la suivante : peut-on distinguer ces deux opérations sur une trace de mesure obtenue via un canal auxiliaire ? Si oui, la séquence des opérations de carrés et de multiplications modulaires conduit à retrouver tout ou partie de l'exposant, qui est secret dans le cas du déchiffrement RSA.

La réponse est donnée par Kocher *et al.* dans [39]. Dans le paragraphe 3 de leur article, ils reproduisent une trace de mesure de puissance électrique consommée lors d'un calcul de déchiffrement RSA (une exponentiation modulaire, donc) que nous montrons figure 3.3. L'algorithme de calcul est ici le *Left-to-right Square-and-multiply* que nous avons rappelé dans l'algorithme 2.1, page 53.

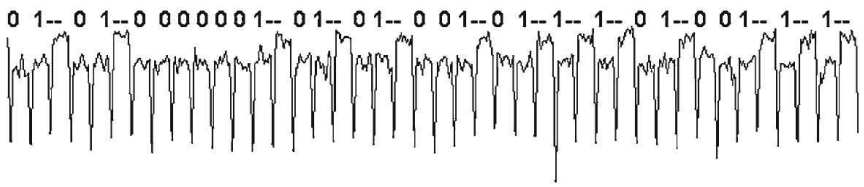


FIGURE 3.3 – Fuite d'information par *Simple Power Analysis*, exponentiation rapide RSA, l'exposant étant la clé secrète, reproduite de [39].

Kocher *et al.* dans [39] ne précisent pas le type de l'appareil mesuré figure 3.3, mais d'une façon générale, la multiplication est plus complexe et plus gourmande en énergie que l'élévation au carré. La trace permet ainsi assez clairement de visualiser les tours de boucle, et d'identifier les opérations respectives (carré ou multiplication) et, par voie de conséquence, d'identifier les bits successifs de k.

Ce cas est un peu extrême mais réaliste : les informations de consommation d'énergie délivrent directement des informations sur la clé (ici l'exposant, qui est la clé secrète RSA).

Dans le cas d'un produit scalaire de point de courbe elliptique effectué avec un algorithme de type *Left-to-right Double-and-add* (voir l'algorithme 2.4, page 68), les opérations précédentes d'élévation au carré et de multiplication modulaire sont remplacées respectivement par le doublement et l'addition de points de la courbe elliptique. Nous avons vu en particulier que les algorithmes et les complexités de ces opérations sont également très différentes entre les deux

(le lecteur peut se reporter aux sections 2.3.2 et 2.3.3). Il apparaît que la multiplication scalaire de point de courbe elliptique laisse fuiter le même genre d'informations.

3.2.2 Principales contre-mesures

3.2.2.1 *Double-and-add/Square-and-multiply always*

En premier lieu, Coron dans [16] propose de modifier l'algorithme 2.4 (page 68) en *Left-to-right Double-and-add-always* (voir l'algorithme 3.1, nous ne présentons pas la variante *Right-to-left*, qui utilise le même procédé). L'idée est d'effectuer des opérations inutiles, précisément de forcer le calcul d'une addition à chaque tour de boucle. Cet algorithme donne aussi un temps de calcul constant quel que soit le jeu de données. En termes de complexité, pour un scalaire k de taille t bits, cet algorithme nécessite :

— t doublements de point ;
— t additions de points.

Algorithme 3.1 *Left-to-right Double-and-add-always*

Require: $k = (k_{t-1}, \ldots, k_1, k_0), P \in E(\mathbb{F}_q)$.
Ensure: $Q = k \cdot P$
1: $Q_0 \leftarrow \mathcal{O}, Q_1 \leftarrow \mathcal{O}$
2: **for** i from $t-1$ downto 0 **do**
3: $\quad Q_0 \leftarrow 2 \cdot Q_0$
4: $\quad Q_1 \leftarrow Q_0 + P$
5: $\quad Q_0 \leftarrow Q_{k_i}$
6: **end for**
7: **return** (Q_0)

Cette contre-mesure s'applique également au cas de l'exponentiation rapide. Dans ce cas, l'algorithme devient le *Left-to-right Square-and-multiply-always* (voir l'algorithme 3.2.) En termes de complexité, pour un exposant e de taille t bits, cet algorithme nécessite :

— t élévations au carré modulaires ;
— t multiplications modulaires.

Algorithme 3.2 *Left-to-right Square-and-multiply-always*

Require: N le module RSA, $e = (e_{t-1}, \ldots, e_1, e_0), g \in \{1, \ldots, N-1\}$.
Ensure: $X = g^e \bmod N$
1: $X_0 \leftarrow 1, X_1 \leftarrow 1$
2: **for** i from $t-1$ downto 0 **do**
3: $\quad X_0 \leftarrow X_0^2 \bmod N$
4: $\quad X_1 \leftarrow X_0 \cdot g \bmod N$
5: $\quad X_0 \leftarrow X_{e_i}$
6: **end for**
7: **return** (X_0)

Hélas, Yen et Joye dans [32] relèvent que cette contre-mesure rend l'opération plus vulnérable à un autre type d'attaque, l'attaque active dite *safe error*, car une erreur injectée avant une des opérations inutiles se traduira par un résultat tout de même correct à la fin, et erroné dans le cas contraire. Une modification d'un bit de Q_1, en particulier, ne donne un résultat faux à la fin que si le bit correspondant de k est 1. Ceci permet à un adversaire de reconstituer l'exposant par essais successifs et injections d'erreurs à des moments différents du calcul. Cette

attaque *safe error* nécessite néanmoins une puissance très supérieure de la part de l'attaquant par rapport à notre SPA, comme nous l'avons vu plus haut.

Pour en finir avec cet algorithme *Left-to-right Double-and-add always*, on déplore une perte d'efficacité non négligeable avec ce type de construction.

3.2.2.2 Exponentiation *Square-always*

Clavier *et al.* dans [15] suggèrent une autre voie pour rendre le calcul de l'exponentiation modulaire régulier. Ils remarquent que toute multiplication peut se récrire de deux manières différentes, comme suit :

$$x \times y = \frac{(x+y)^2 - x^2 - y^2}{2} \tag{3.1}$$

et

$$x \times y = \left(\frac{x+y}{2}\right)^2 - \left(\frac{x-y}{2}\right)^2 \tag{3.2}$$

Clavier *et al.* remarquent en particulier que, dans ces formules, la division modulaire par deux d'un élément A de l'anneau $\mathbb{Z}/N\mathbb{Z}$ s'effectue de façon efficace en appliquant la méthode suivante :

$t_0 \leftarrow A$
$t_1 \leftarrow A + N$
$\alpha \leftarrow A \bmod 2$
return $t_\alpha/2$

Ceci permet de reconstruire l'algorithme *Square-and-multiply* de façon à n'employer que des élévations au carré : l'algorithme 3.3 *Square-always*. Nous présentons cet algorithme en version *Left-to-right*, reconstruit avec l'équation (3.2). Néanmoins, l'utilisation de l'équation (3.1) est équivalente, en remarquant que, dans l'équation (3.1), le calcul de $X^2 \bmod N$ étape 3 de l'algorithme peut réutiliser celui effectué lors de l'itération précédente de la boucle pour l'évaluation de $X \leftarrow X \cdot g \bmod N$.

En conclusion, quelle que soit la méthode utilisée, la complexité de l'algorithme 3.3 correspond en moyenne à deux élévations au carré par tour de boucle, soit pour un exposant de taille t bits, $2t$ élévations au carré pour l'exponentiation modulaire complète. Il convient cependant d'y ajouter en moyenne $1,5t$ additions modulaires et t divisions par deux, avec l'équation (3.2), selon la méthode décrite plus haut. Clavier *et al.* dans [15] donnent également la version *Right-to-left* de cette approche, dont la complexité est strictement identique.

En l'état, cette approche n'est pas totalement régulière en raison des additions. Clavier *et al.* dans [15] proposent donc une version atomique de la boucle principale de l'algorithme 3.3. Dans cette version que nous ne présentons pas ici, le test **if** est supprimé et le corps de la boucle comporte une séquence unique d'opérations qui utilise des registres pour stocker des éléments intermédiaires. Le lecteur peut se reporter à Clavier *et al.* dans [15] pour prendre connaissance de ces approches atomiques, dont la complexité est inchangée par rapport à notre analyse précédente.

On remarque pour finir que le temps d'exécution de cet algorithme dépend du poids de Hamming du scalaire traité, en raison du test **if** de l'étape 4 de l'algorithme 3.3. Ceci permet à un attaquant de connaître cette information par chronométrage du temps d'exécution.

3.2.2.3 Échelle binaire de Montgomery, amélioration dans le cas des courbes elliptiques sur corps finis

Pour améliorer les parades précédentes, une solution classique maintenant dans la littérature est d'utiliser l'échelle binaire de Montgomery, dont la première présentation par Mont-

Algorithme 3.3 *Left-to-right Square-always*, d'après Clavier *et al.* dans [15]

Require: Les entiers g et $e = (e_{t-1}, \dots e_0)_2 \in [1, \dots, N[$, avec $e_{t-1} = 1$.
Ensure: $X = g^e \bmod N$

```
1: X ← g
2: for i = t − 2 downto 0 do
3:    X ← X^2 mod N
4:    if e_i = 1 then
5:       T_0 ← ((X+g)/2)^2 mod N
6:       T_1 ← ((X−g)/2)^2 mod N
7:       X ← T_0 − T_1 mod N
8:    end if
9: end for
```

gomery se trouve dans [47]. La forme basique de cette échelle est reproduite dans les algorithmes 3.4 et 3.5, respectivement pour la multiplication scalaire de point de courbe elliptique et l'exponentiation modulaire. Dans cette première version, on effectue toujours un doublement et une addition à chaque tour de boucle (respectivement un carré et une multiplication). L'efficacité est donc toujours inférieure à ce que l'on a avec l'algorithme classique.

L'observation de l'algorithme 3.4 permet de remarquer une propriété intéressante : la différence $Q_1 - Q_0 = P$ est conservée durant la totalité du calcul. On le montre par la récurrence suivante :

- étape 1 de l'algorithme 3.4, on a bien initialement $Q_1 - Q_0 = 2P - P = P$
- on suppose que $Q_1 - Q_0 = P$ est vérifié pour le bit $i + 1$ de la boucle principale ; on a alors
  - si le bit du scalaire $k_i = 0$, alors $(Q_0 + Q_1) - 2Q_0 = Q_1 - Q_0 = P$;
  - si le bit du scalaire $k_i = 1$, alors $2Q_1 - (Q_0 + Q_1) = Q_1 - Q_0 = P$;
  - la propriété $Q_1 - Q_0 = P$ est donc vérifiée également au bit i.

Ceci permet de répondre aux critiques de vulnérabilité de la parade *Double-and-add-always* face aux attaques de type *safe error*. En effet, les dépendances entre les itérations de la boucle entre les valeurs successives de Q_0 et Q_1 rendent toutes les opérations nécessaires à l'élaboration du résultat final. Un calcul erroné d'une coordonnée de Q_0 ou de Q_1 à un moment ou à un autre compromet le résultat final et par conséquent, une injection de faute pourra être vérifiée à tout moment, aucun calcul effectué n'étant inutile.

L'algorithme 3.4 est présenté pour un groupe additif, et on peut de la même manière construire un algorithme d'exponentiation pour un groupe multiplicatif. C'est l'algorithme 3.5. La propriété $Q_1 - Q_0 = P$ dans l'algorithme 3.4 est vérifiée sous la forme $X_1/X_0 \equiv g \bmod N$ dans le cas de l'algorithme 3.5.

En termes de complexité, la situation est la suivante :

- pour l'algorithme 3.4, la multiplication scalaire de point de courbe elliptique :
  - t doublements de points ;
  - t additions de points.
- pour pour l'algorithme 3.5, l'exponentiation modulaire :
  - t élévations au carré modulaires ;
  - t multiplications modulaires ;

La situation est donc la même que pour la contre-mesure précédente *Double-and-add-always*.

Algorithme 3.4 Échelle binaire de Montgomery pour la multiplication scalaire de point de courbe elliptique

Require: $k = (k_{t-1}, \ldots, k_1, k_0)$ avec $k_{t-1} = 1, P \in E(\mathbb{F}_q)$.
Ensure: $Q = k \cdot P$
1: $Q_0 \leftarrow P, Q_1 \leftarrow 2P$
2: **for** i from $t-2$ downto 0 **do**
3: **if** $(k_i = 0)$ **then**
4: $Q_1 \leftarrow Q_0 + Q_1, Q_0 \leftarrow 2 \cdot Q_0$
5: **else**
6: $Q_0 \leftarrow Q_0 + Q_1, Q_1 \leftarrow 2 \cdot Q_1$
7: **end if**
8: **end for**
9: **return** (Q_0)

Algorithme 3.5 Échelle binaire de Montgomery pour l'exponentiation rapide

Require: $e = (e_{t-1}, \ldots, e_1, e_0)$ avec $e_{t-1} = 1, N$ le module, $g \in \{1, \ldots, N-1\}$.
Ensure: $X = g^e \bmod N$
1: $X_0 \leftarrow g, X_1 \leftarrow g^2 \bmod N$
2: **for** i from $t-1$ downto 0 **do**
3: **if** $(e_i = 0)$ **then**
4: $X_1 \leftarrow X_0 \cdot X_1 \bmod N, X_0 \leftarrow X_0^2 \bmod N$
5: **else**
6: $X_0 \leftarrow X_0 \cdot X_1 \bmod N, X_1 \leftarrow X_1^2 \bmod N$
7: **end if**
8: **end for**
9: **return** (X_0)

3.2.2.3.1 Cas particulier de la multiplication scalaire de point de courbe elliptique

Une amélioration de l'échelle binaire de Montgomery est possible dans ce cas, proposée par Montgomery dans [47], à l'origine pour des courbes d'un type particulier appelées depuis courbes de Montgomery. En effet, la propriété $Q_1 - Q_0 = P$ au long du calcul vue plus haut permet d'effectuer les calculs uniquement sur la coordonnée x des deux points de la courbe considérée, dans le cas de coordonnées affines. On parle dans ce cas de pseudo-addition. Lopez et Dahab dans [43] ont adapté cette variante au cas des corps binaires. Hankerson *et al.* dans [22] présentent ce dernier algorithme pour les coordonnées projectives, qui offrent l'avantage d'économiser un grand nombre d'inversions. Nous transcrivons en algorithme 3.6 cette variante sur le corps $\mathbb{F}_{2^m}$.

Algorithme 3.6 Échelle binaire de Montgomery pour coordonnées projectives, courbe $E(\mathbb{F}_{2^m})$

Require: $k = (k_{t-1}, \ldots, k_1, k_0)$ avec $k_{t-1} = 1, P(x, y) \in E(\mathbb{F}_{2^m})$.
Ensure: $Q = k \cdot P$
1: $X_1 \leftarrow x, Z_1 \leftarrow 1, X_2 \leftarrow x^4 + b, Z_2 \leftarrow x^2$
2: **for** i from $t-2$ downto 0 **do**
3: **if** $(k_i = 0)$ **then**
4: $T \leftarrow Z_1, Z_1 \leftarrow (X_1 Z_2 + X_2 Z_1)^2, X_1 \leftarrow xZ_1 + X_1 X_2 T Z_2$
5: $T \leftarrow X_2, X_2 \leftarrow X_2^4 + bZ_2^4, Z_2 \leftarrow T^2 + Z_2^2$
6: **else**
7: $T \leftarrow Z_2, Z_2 \leftarrow (X_1 Z_2 + X_2 Z_1)^2, X_2 \leftarrow xZ_2 + X_1 X_2 T Z_1$
8: $T \leftarrow X_1, X_1 \leftarrow X_1^4 + bZ_1^4, Z_1 \leftarrow T^2 + Z_1^2$
9: **end if**
10: **end for**
11: $x_3 \leftarrow X_1/Z_1$
12: $y_3 \leftarrow (x + x_3)[(X_1 + xZ_1)(X_2 + xZ_2) + Z_1 Z_2(x^2 + y)](xZ_1 Z_2)^{-1} + y$
13: **return** $(Q(x_3, y_3))$

L'intérêt de ce dernier algorithme est la complexité inférieure qu'il présente. En effet, on note que :

— étapes 4 ou 7, on a pour complexité $5M + 1S$;
— étapes 5 ou 8, on effectue $1M + 5S$;

soit au total $6M + 6S$ par tour de boucle, à comparer au coût de l'addition et du doublement de points : $13M + 8S$ (voir la table 2.12 page 67). Sous cette forme, l'amélioration de performance de cette contre-mesure est ici substantielle.

Enfin, on note que cette amélioration s'applique aussi aux courbes sur corps finis de grande caractéristique $\mathbb{F}_p$. Les formules de pseudo-additions en coordonnées projectives (coordonnées XZ) peuvent être trouvées dans [3].

3.2.2.4 Multiplication scalaire hautement régulière *Double-add*

Joye dans [30] propose un algorithme qui répond à la critique des précédents sur la faiblesse face à l'attaque *safe error*. Cet algorithme est construit sur la remarque que nous présentons maintenant.

Soit $k = \sum_{i=0}^{t-1} k_i 2^i$ avec $k_i \in \{0, 1\}$. On a alors

$$Q = kP = \sum_{i=0}^{t-1} (k_i 2^i) P = \sum_{i=0}^{t-1} k_i B_i \text{ avec } B_i = 2^i P.$$

On définit maintenant

$$S_j = \sum_{i=0}^{j} k_i B_i \text{ et } T_j = B_{j+1} - S_j.$$

On a alors

$$\begin{aligned} S_j &= \sum_{i=0}^{j} k_i B_i = k_j B_j + S_{j-1} = k_j(S_{j-1} + T_{j-1}) + S_{j-1} \\ &= (1 + k_j)S_{j-1} + k_j T_{j-1}. \end{aligned}$$

et

$$\begin{aligned} T_j &= B_{j+1} - S_j = 2B_j - (k_j B_j + S_{j-1}) = (2 - k_j)B_j - S_{j-1} \\ &= (2 - k_j)T_{j-1} + (1 - k_j)S_{j-1}. \end{aligned}$$

De façon équivalente, Joye montre que, pour tout $j \geq 0$,

$$S_j = \begin{cases} S_{j-1} & \text{si } k_j = 0 \\ 2S_{j-1} + T_{j-1} & \text{si } k_j = 1 \end{cases} \quad \text{et} \quad T_j = \begin{cases} S_{j-1} + 2T_{j-1} & \text{si } k_j = 0 \\ T_{j-1} & \text{si } k_j = 1 \end{cases}.$$

Joye construit donc un algorithme régulier dans lequel aucune opération n'est inutile. C'est l'algorithme 3.7, baptisé *Double-add*.

Algorithme 3.7 Multiplication scalaire de point de courbe elliptique *Double-add*, Joye [30]

Require: $k = (k_{t-1}, \ldots, k_1, k_0)$ avec $k_{t-1} = 1, P \in E(\mathbb{F}_q)$.
Ensure: $Q = k \cdot P$
1: $Q_0 \leftarrow \mathcal{O}, Q_1 \leftarrow P$
2: **for** i from 0 to $t - 1$ **do**
3: $\quad b \leftarrow 1 - k_j, Q_b \leftarrow 2Q_b + Q_{k_j}$
4: **end for**
5: **return** (Q_0)

Joye, toujours dans [30], développe à partir de l'algorithme 3.7 *Double-add* quelques variantes que nous ne présentons pas, mais qui conservent les mêmes propriétés et peuvent améliorer l'efficacité dans certains cas. La complexité de cette approche est en effet la même que dans le cas précédent. Pour un scalaire k de taille t bits, cet algorithme nécessite :

— t doublements de points ;
— t additions de points.

On note pour finir qu'il est possible d'adapter cette méthode à l'exponentiation modulaire.

3.2.2.5 Exponentiation régulière 2^γ-ary

Cet algorithme a été proposé par Joye et Tunstall dans [31], et s'applique également à la multiplication scalaire de point de courbe elliptique. Il est résistant à l'attaque *Simple Power Analysis*.

Pour gagner en efficacité, l'utilisation de l'exponentiation 2^γ-ary permet de diminuer le nombre de multiplications de façon significative. Cette méthode calcule l'exponentiation de la façon suivante : on utilise la représentation $(e_{\ell-1}, \ldots, e_0)_\beta$ en base $\beta = 2^\gamma$ de e. Pour une taille binaire t de l'exposant $e \in \mathbb{N}$ (on a donc $2^{t-1} \leq e < 2^t$), la taille de e est $\ell = \lceil t/\gamma \rceil$ en base $\beta = 2^\gamma$ et on calcule

$$g^e = \prod_{i=0}^{\ell} g^{\beta^i e_i} \tag{3.3}$$

La table suivante donne la comparaison avec le classique *Square-and-multiply* (voir l'algorithme 2.1 page 53) :

Exponentiation	*Binary Square-and-multiply*	*β-ary Square-and-multiply*
# carrés	t	$\lfloor t/\gamma \rfloor \times \gamma + 1$
# moyen de multiplications	$t/2$	$\ell = \lceil t/\gamma \rceil + 2^\gamma$

Les complexités sont données dans le cas *left-to-right*.

En base deux pour un exposant aléatoire, la proportion de bits nuls est la moitié en moyenne. Dans la représentation β-aire, la proportion de zéros est de $1/\beta$. L'idée de Joye et Tunstall est de modifier le représentation β-aire pour éliminer les zéros, en utilisant les chiffres dans l'ensemble $\{1, \ldots, \beta\}$.

Le but est de calculer $(X = g^e)$ avec la représentation β-aire de $e = \sum_{i=0}^{\ell-1} e_i \beta^i$. Joye et Tunstall modifient cette écriture de la façon suivante : on considère $s = \sum_{i=0}^{\ell-1} \beta^i$ et on définit $e' = e - s$. Si $e' = \sum_{i=0}^{\ell-1} e'_i \beta^i$, on a alors

$$\begin{aligned} g^e &= g^{e'+s} \\ &= g^{\sum_{i=0}^{\ell-1} \kappa_i \beta^i} \quad \text{avec } \kappa_i = e'_i + 1. \end{aligned}$$

L'algorithme 3.8 met en œuvre cette méthode de réécriture du scalaire.

Algorithme 3.8 Réécriture de Joye-Tunstall

Require: $e \geq 1$, $\beta = 2^\gamma$, ℓ la longueur β-aire de e et le module RSA N.
Ensure: $e = (e_{\ell-1}, \ldots, e_0)$ avec $e_i \in \{1, \ldots, \beta\}$, $1 \leq i \leq \ell - 2$
1: $s \leftarrow (1, 1, \ldots, 1)_\beta$
2: $e \leftarrow e - s \bmod \phi(N)$
3: **for** $i = 0$ to $\ell - 2$ **do**
4: $\quad d \leftarrow e \bmod \beta$
5: $\quad e \leftarrow \lfloor e/\beta \rfloor$
6: $\quad e_i \leftarrow d + 1$
7: **end for**

Exemple 3.2.1. *Un exemple en décimal ($t = 4, \beta = 10_{10}$) permet d'illustrer cet algorithme. L'entier e est tiré au sort sur 9 chiffres en base 10. On effectue en premier lieu la soustraction $e - s \bmod N$ de l'étape 2 de l'algorithme 3.8.*

e	$=$	1	9	7	5	0	4	0	2	3
s	$=$	1	1	1	1	1	1	1	1	1
$e' \leftarrow e - s \bmod N$		0	8	6	3	9	2	9	1	2

Alors, l'addition sans retenue $e' + s$ conduit à la représentation sans «0»

$e' \bmod N$		0	8	6	3	9	2	9	1	2
s	$=$	1	1	1	1	1	1	1	1	1
recodage		1	9	7	4	10	3	10	2	3

On constate bien dans notre exemple que les zéros ont été remplacés par des 10_{10}, et que les chiffres suivants sont dimimués de la retenue correspondante, comme on le faisait à l'école primaire pour faire des soustractions.

L'utilisation de cette représentation, en éliminant les chiffres nuls, rend l'algorithme d'exponentiation β-aire régulier et de ce fait résistant à l'attaque SPA. Cette méthode est montrée dans ses deux versions *Left-to-right* et *Right-to-left*, respectivement les algorithmes 3.9 et 3.10.

Algorithme 3.9 *Regular 2^γ-ary Left-to-right Square-and-multiply*

Require: g, réécriture de Joye-Tunstall de $e = (e_{\ell-1}, \dots e_0) < N$.
Ensure: $X = g^e \bmod N$

1: $Y_1 \leftarrow g$
2: **for** $i = 2$ to 2^γ **do**
3: $\quad Y_i \leftarrow Y_{i-1} \cdot g \bmod N$
4: **end for**
5: $X \leftarrow Y_{e_\ell - 1}$
6: **for** $i = \ell - 2$ downto 0 **do**
7: $\quad X \leftarrow X^{2^\gamma} \bmod N$
8: $\quad X \leftarrow X \cdot Y_{e_i} \bmod N$
9: **end for**
10: **return** $(X = g^e)$

Algorithme 3.10 *Regular 2^γ-ary Right-to-left Square-and-multiply*

Require: g, réécriture de Joye-Tunstall de $e = (e_{\ell-1}, \dots e_0) < N$.
Ensure: $X = g^e \bmod N$

1: $X \leftarrow g, Y_i \leftarrow 1, i \in \{1, \dots, 2^\gamma\}$
2: **for** $i = 0$ to $\ell - 1$ **do**
3: $\quad Y_{e_i} \leftarrow Y_{e_i} \cdot X \bmod N$
4: $\quad X \leftarrow X^{2^\gamma} \bmod N$
5: **end for**// Reconstruction finale
6: $X \leftarrow Y_{2^\gamma}$
7: **for** $i = 2^\gamma - 1$ downto 1 **do**
8: $\quad Y_i \leftarrow Y_i \cdot Y_{i+1} \bmod N$
9: $\quad X \leftarrow X \cdot Y_i \bmod N$
10: **end for**
11: **return** $(X = g^e)$

Cette contre-mesure est très intéressante, car elle allie efficacité (on diminue le nombre de multiplications et de carrés modulaires) tout en apportant une régularité et en présentant une séquence d'opérations indépendante de la valeur de l'exposant e.

3.2.2.6 Bilan comparatif des complexités des contre-mesures

Un bilan sous forme d'un tableau synthétise les complexités de calcul. La comparaison porte sur le nombre d'opérations réalisées en fonction de t, la longueur binaire du scalaire k pour les courbes elliptiques sur $\mathbb{F}_{2^m}$. Les résultats que nous montrons ici dans la table 3.1 sont issus de la compilation des données de chaque algorithme pour les multiplications scalaires et des résultats montrés dans la table 2.12 page 67 pour les opérations sur points de courbe. On remarque que dans cette table, pour la valeur de γ nous avons retenu :

— pour la représentation γ-NAF, $\gamma = 4$;
— pour la base $\beta = 2^\gamma$ dans les approches *Regular Left-to-right/Right-to-left Double-and-add*, algorithmes 3.9 et 3.10, $\gamma = 5$.

En effet, ces deux valeurs sont celles couramment admises comme les meilleures dans la littérature.

Algorithme	Coord.	Points stockés	Opérations sur EC Add.	Dbl.	Opérations sur le corps Total en multiplication
Double-and-add	aff.	0	$\lceil t/2 \rceil$	t	$\approx 19, 2t$
algorithme 2.4	proj.	0	$\lceil t/2 \rceil$	t	$\approx 13, 3t + 10$
γ-*NAF Double-and-add*	proj.	0	$\lceil t/5 \rceil + 4$	t	$\approx 9, 7t + 60, 4$
algorithme 2.8 $\gamma = 4$					
Double-and-add-always	aff.	0	t	t	$25, 6t$
algorithme 3.1	proj.	0	t	t	$19, 8t + 10$
Double-add	aff.	0	t	t	$25, 6t$
algorithme 3.7	proj.	0	t	t	$19, 8t + 10$
Montgomery algo. 3.6	proj.	0	t	t	$10, 8t + 30, 6$
Reg. L-R D.&A. algo. 3.9	proj.	32	$\lceil t/5 \rceil + 32$	$\lfloor t/5 \rfloor \times 5 + 1$	$\approx 9, 7t + 420, 4$
Reg. R-L D.&A. algo. 3.10	proj.	32	$\lceil t/5 \rceil + 64$	$\lfloor t/5 \rfloor \times 5 + 1$	$\approx 9, 7t + 823, 6$
$\gamma = 5$					

Sur le corps $\mathbb{F}_{2^m}$, on fait l'hypothèse qu'une inversion coûte 10 multiplications, et que S = 0,8M

TABLE 3.1 – Bilan de complexité pour les différents algorithmes de multiplication scalaire *SPA resistant*, sur $E(\mathbb{F}_{2^m})$ pour un scalaire de longueur t bits.

Pour ce qui concerne l'exponentiation, nous synthétisons dans la table 3.2 les complexités correspondantes. De même que précédemment, nous avons retenu $\gamma = 5$ pour les approches *Regular Left-to-right/Right-to-left Square-and-multiply*, algorithmes 3.9 et 3.10.

Algorithme	stockage	mult.	carrés	Total en M
Square-and-multiply algorithme 2.2 ou 2.1	0	$\lceil t/2 \rceil$	t	$1, 3t$
Square-and-multiply-Always algorithme 3.2	0	t	t	$1, 8t$
Square-always algorithme 3.3	0	0	$2t$	$1, 6t$
Montgomery algo. 3.6	0	t	t	$1, 8t$
Reg. L-R S.-and-M. algo. 3.9	32	$\lceil t/5 \rceil + 32$	$\lfloor t/5 \rfloor \times 5 + 1$	$\approx t + 33$
Reg. R-L S.-and-M. algo. 3.10	32	$\lceil t/5 \rceil + 64$	$\lfloor t/5 \rfloor \times 5 + 1$	$\approx t + 65$
$\gamma = 5$				

On considère ici S = 0,8M

TABLE 3.2 – Bilan de complexité pour les différents algorithmes d'exponentiation modulaire *SPA resistant*, dans anneau $\mathbb{Z}/N\mathbb{Z}$ avec un exposant de taille t bits.

3.3 *Timing Attack*

Le temps d'exécution d'un calcul d'exponentiation modulaire ou d'une multiplication scalaire de point de courbe elliptique est une donnée facilement accessible pour un adversaire. En effet, dans un protocole interactif, un adversaire peut solliciter une cible en envoyant un message à traiter. Il chronomètre alors simplement le temps mis pour obtenir une réponse. Dans un algorithme non régulier, le temps de calcul peut varier en fonction du poids de Hamming de l'exposant ou du scalaire. Ceci donne une information à l'attaquant. Cependant, cela reste

insuffisant pour retrouver la totalité de l'exposant ou du scalaire. Un lot de chronométrages de calculs avec plusieurs jeux de données différents, mais avec le même exposant ou scalaire secret, peut présenter des propriétés statistiques qui révèlent plus d'informations sur l'exposant ou le scalaire. Ainsi, et contrairement à la SPA précédente, il faudra obtenir plusieurs mesures. Cette attaque repose donc sur un principe statistique : la distribution des temps présente des caractéristiques (variance) dépendantes de l'implantation et des messages à traiter, mais qui révèlent l'exposant ou le scalaire secret convoité.

Cette attaque a été présentée par Kocher dans [37]. Nous reprenons sa présentation avec en premier lieu une attaque simplifiée, puis la version générale de l'attaque. Nous conclurons sur l'application et les contre-mesures de cette attaque.

3.3.1 Description de l'attaque

Kocher dans [37] se place dans le cas de l'exponentiation modulaire que nous avons présentée en algorithme 2.1 page 53. Nous reprenons les notations de l'algorithme 2.1 avec en plus : pour $b < t$, X_b est la valeur obtenue après b tours de boucle (on a donc $g^e = X_{t-1}$).

3.3.1.1 Attaque simplifiée

Dans un premier temps, Kocher suppose que le temps d'exécution de la multiplication modulaire dépend des données en entrée, et que l'attaquant connaît les données pour lesquelles ce temps est anormalement long. Dans cette hypothèse extrême, l'attaquant qui a recueilli plusieurs temps et qui connaît les premiers bits de l'exposant peut deviner le bit suivant. En procédant de façon itérative, la totalité de l'exposant peut être connue.

On suppose donc que l'attaquant connaît les b premiers bits de l'exposant. Il peut calculer les différentes valeurs de X_b correspondant à chaque temps qu'il a obtenu après avoir sollicité différents calculs de la cible. À l'itération suivante, l'algorithme calcule $Y = X_b^2$, puis $Y \cdot g \bmod N$ si le $(b+1)^{\text{ème}}$ bit vaut 1 ($e_{t-b-1} = 1$). Deux cas se présentent :

— soit l'hypothèse $e_{t-b-1} = 1$ est vraie : la multiplication $Y \cdot g \bmod N$ est donc calculée par l'algorithme. Dans certains cas que l'attaquant peut déterminer, cette multiplication va prendre un temps plus long. Il peut alors vérifier que c'est le cas des exécutions correspondantes ;

— soit l'hypothèse $e_{t-b-1} = 0$ est vraie : la multiplication n'est jamais effectuée, l'attaquant peut alors voir que les exécutions lentes n'apparaissent pas de façon corrélée ;

— enfin, l'attaquant peut vérifier l'hypothèse $e_{t-b-1} = 1$ ou $e_{t-b-1} = 0$ en contrôlant le temps d'exécution du carré suivant, selon les données en entrée par rapport au temps d'exécution de ce carré qu'il prédit.

Cette version simplifiée de l'attaque est peu réaliste. Elle suppose une propriété particulière de la multiplication modulaire, soit un temps d'exécution dépendant du jeu de données d'entrée, au moins pour certains d'entre eux, que l'attaquant connaît. Ceci dit, le principe est posé d'une attaque que l'on va raffiner sur le plan statistique pour en renforcer l'efficacité.

3.3.1.2 Attaque générale

Soient m messages $y_0, y_1, \ldots, y_{j-1}$ et les temps de calcul $T_0, T_1, \ldots, T_{j-1}$. Kocher dans [37] remarque que chaque temps de calcul peut s'écrire $T = \epsilon + \sum_{i=0}^{t-1} \tau_i$ où τ_i est le temps d'exécution de chacune des itérations i de la boucle `for` de l'algorithme 2.1 et ϵ les erreurs et bruit de mesure. Connaissant les b premiers bits de l'exposant e, deviner e_b peut s'effectuer avec une bonne probabilité de la façon suivante :

— l'attaquant peut calculer $\sum_{i=0}^{b-1} \tau_i$ pour chaque message $y_0, y_1, \ldots, y_{j-1}$;

— les temps T_j étant indépendants les uns des autres, il peut calculer la variance des $\theta_j = \epsilon + \sum_{i=b}^{t-1} \tau_i = T_j - \sum_{i=0}^{b-1} \tau_i$ pour chacun des j messages ;
— l'attaquant formule une hypothèse sur e_b, il a deux alternatives, 0 ou 1 ; l'attaquant peut ensuite calculer τ_b et $\sum_{i=0}^{b} \tau_i$, et après avoir effectué ce calcul pour les j messages, en déduire la nouvelle valeur de la variance de $\theta_j = T_j - \sum_{i=0}^{b} \tau_i$:
  — si l'hypothèse sur e_b est correcte, la valeur attendue de la variance correspond à celle que l'on calcule directement à partir de celle du bruit et des itérations de la boucle avec les bonnes hypothèses sur les bits. Cette valeur est

$$\text{Var}(\epsilon) + (t-b)\text{Var}(\tau);$$

  — si l'hypothèse sur e_b est fausse, les valeurs réelles de τ_b attendues par l'attaquant sont décorrélées de son hypothèse, et on accroît donc la variance attendue par rapport à une hypothèse correcte. La variance attendue devient donc

$$\text{Var}(\epsilon) + (t-b+2)\text{Var}(\tau).$$

L'attaquant réitère ce processus pour chaque bit. Le succès de l'attaque est assuré par le fait que $\text{Var}(\epsilon)$ n'est pas grand devant $(t-b)\text{Var}(\tau)$ et que chaque hypothèse successive correcte sur les bits de l'exposant diminue la variance de θ_j de $\text{Var}(\tau)$, alors qu'une hypothèse incorrecte l'augmente de $\text{Var}(\tau)$. Ainsi, plus simplement, la bonne hypothèse est celle qui produit la variance de θ_j la plus petite.

Kocher tire aussi de cette dernière propriété le nombre de chronométrages requis pour assurer le succès de l'attaque. Il évalue pour ce faire la probabilité de vérifier la bonne hypothèse sur le bit b en négligeant $\text{Var}(\epsilon)$, c'est à dire que les variances mentionnées plus haut sont correctement ordonnées. Il utilise pour ce faire l'hypothèse que les valeurs τ_i suivent une loi normale. On a dans ce cas :

$$\mathcal{P}_b = P\left(\sum_{i=0}^{j-1}\left(\sqrt{t-b}X_i + \sqrt{2}Y_i\right)^2 > \sum_{i=0}^{j-1}\left(\sqrt{t-b}X_i\right)^2\right),$$

où X et Y sont deux variables aléatoires standards suivant la loi normale (les moyennes de X et Y sont 0 et leur écart-type est 1). Après développements et quelques réarrangements, cette probabilité s'écrit :

$$\mathcal{P}_b = P\left(Z > -\frac{\sqrt{j}}{\sqrt{2(t-b)}}\right),$$

où Z est une variable aléatoire suivant la loi normale dont la moyenne est 0 et l'écart-type est 1. On écrit aussi

$$\mathcal{P}_b = \Phi\left(\sqrt{\frac{j}{2(t-b)}}\right).$$

Φ représente l'aire sous la courbe de Gauss représentant Z entre $-\infty$ et son argument, ici $\sqrt{\frac{j}{2(t-b)}}$.

Exemple 3.3.1. *Nous calculons la probabilité d'avoir la bonne hypothèse sur le bit b dans les conditions suivantes :*

— *le nombre de calculs chronométrés est $j = 1000$;*
— *la taille de l'exposant est 1024, et c'est aussi le nombre d'itérations de la boucle t ;*

— *on veut deviner le premier bit* $b = 1$;
alors, la probabilité de succès dans l'hypothèse réalisée est

$$\mathcal{P}_b = \Phi\left(\sqrt{\frac{1000 \times 1}{2(1024-1)}}\right) \approx 0,7580.$$

Dans notre exemple, la taille correspond à ce qui est préconisé pour une signature. On remarque aussi que si une hypothèse est fausse, il est toujours possible de le constater car dans ce cas, la variance, qui doit diminuer au fur et à mesure des hypothèses, augmente au contraire. Ainsi, une hypothèse incorrecte peut être détectée et l'attaque ajustée en conséquence.

3.3.2 Contre-mesures et conclusion

La *Timing Attack* s'appuie sur le manque de régularité des opérations internes à la boucle `for` des algorithmes d'exponentiation modulaire ou de multiplication scalaire de point de courbe elliptique. Ce manque de régularité peut avoir deux causes : soit les opérations réalisées dépendent du bit du scalaire (comme dans l'algorithme 2.1 pour l'exponentiation modulaire), soit en raison d'une dépendance du temps de calcul d'une opération élémentaire en fonction du jeu de données d'entrée de cette opération.

La parade consiste donc à utiliser des opérations régulières autant que faire se peut afin de rendre la variance des erreurs de mesure du bruit $\mathrm{Var}(\epsilon)$ aussi grande que possible devant le terme $(t-b)\mathrm{Var}(\tau)$ de chaque tour de boucle.

Kocher pense cependant que l'utilisation d'opérations dont la séquence d'instructions est indépendante des données d'entrée telles que la multiplication modulaire de Montgomery dans l'exponentiation modulaire (voir l'algorithmes 1.3 à 1.8 pages 25 à 30) n'est pas une contre-mesure suffisante si le niveau de bruit reste acceptable. Il ne donne cependant pas d'évaluation précise. Le cas de la multiplication scalaire de point de courbe elliptique est aussi menacée, avec la même remarque au niveau de la régularité des opérations de doublement, addition ou *halving* pour les courbes sur corps de caractéristique 2.

Le cas des algorithmes réguliers d'exponentiation ou de multiplication scalaire répond cependant à cette critique. En effet, si les opérations effectuées à chaque tour de boucle sont strictement identiques et indépendantes des données, la variance du temps de calcul τ_i pour chaque tour est constante. Cette régularité ne permet donc pas de distinguer les différents bits. Il ne reste dans la variance que $\mathrm{Var}(\epsilon)$. Ceci reste à prendre avec prudence dans la mesure où une indépendance totale vis à vis des données d'entrée ne peut être garantie pour des raisons qui peuvent relever de l'architecture (défauts de cache, ou l'exemple des tests étape 9 de l'algorithme 1.8 page 30 de débordement dans la réduction modulaire...) Nous pouvons cependant admettre dans la suite que les algorithmes réguliers tels que l'échelle binaire de Montgomery ou les exponentiations régulières 2^γ-aires sont résistantes à ces attaques (les algorithmes 3.5 à 3.6, 3.9 et 3.10).

Pour finir, une contre-mesure plus radicale peut être mise en œuvre : c'est la randomisation de l'exposant/scalaire. Nous examinons quelques-unes de ces techniques plus loin, dans la section 3.4.2 sur les contre-mesures face à l'attaque *Differential Power Analysis*. Dans ce dernier cas, le masquage des bits de l'exposant/scalaire entre chaque calcul ne donne plus aucune signification au calcul de variance précédent.

3.4 DPA : *Differential Power Analysis*

Nous avons conclu sur l'attaque SPA avec sa relative sensibilité aux bruits de mesure, et la difficulté d'interpréter les fuites d'informations sur la base d'une mesure unique ou en tout

petit nombre. Un attaquant peut contourner cette difficulté s'il dispose de plus grands moyens, par exemple, un plus grand nombre d'enregistrements (quelques centaines à quelques milliers) et la connaissance du texte clair ou chiffré selon le cas. Dans cette attaque, l'adversaire a donc une puissance plus importante, mais pas irréaliste.

Dans le cas qui nous intéresse de l'exponentiation modulaire ou de la multiplication scalaire de point de courbe elliptique, Coron dans [16] donne une version adaptée de l'attaque DPA.

Dans cette section, nous présentons cette attaque et donnons en conclusion quelques contre-mesures.

3.4.1 Description de l'attaque

Plaçons nous dans le cas ou la multiplication scalaire est effectuée avec l'approche *Double-and-add* (Voir l'algorithme 2.4 page 68).

On remarque premièrement que les résultats $Q = k \cdot P$ lors de l'itération j de la boucle dépendent des bits traités antérieurement à j, soit $(k_{t-1}, \ldots, k_j)$ si k est codé sur t bits.

Si on considère les premiers tours de boucle, après le premier doublement (étape 3 de l'algorithme 2.4) correspondant au bit le plus significatif de k (k_{t-1}), on s'aperçoit que pour le bit suivant k_{t-2}, l'algorithme va ajouter P si ce bit est à 1 (étape 5 de l'algorithme 2.4) et sinon, n'effectue pas l'addition. Après le premier tour de boucle, le point courant à traiter au tour suivant est donc soit :

— $2P$ pour $k_{t-2} = 0$,

— ou $3P$ pour $k_{t-2} = 1$.

À l'itération suivante, on calcule donc le doublement à partir de ce résultat, soit :

— $4P$ pour $k_{t-2} = 0$,

— ou $6P$ pour $k_{t-2} = 1$.

Le point $4P$ n'est donc calculé que pour $k_{t-2} = 0$. Toute consommation électrique mesurable et dépendante de cette valeur «trahira» donc le bit $k_{t-2} = 0$. Dans le cas contraire, on aura une consommation décorrélée de cette valeur $k_{t-2} = 0$, et on en déduira alors $k_{t-2} = 1$.

L'attaque nécessite de la part de l'adversaire :

— la mesure de la puissance consommée lors de m calculs de multiplication scalaire $\mathbf{T}_{1\ldots m}[j]$;

— La connaissance du point de base utilisé $P_1, P_2, \ldots, P_m$.

La puissance de l'adversaire n'est pas irréaliste comme on le voit : il lui faut connaître soit le texte chiffré, soit le texte clair, correspondant à chacune des mesures effectuées. Le nombre de mesures est de quelques centaines à quelques milliers, et peut être évalué.

Ceci permet de construire la fonction de sélection $D(P_i, s)$ pour diviser les enregistrements en deux sous-ensembles. Soit l'ensemble des m mesures (pour différents points $P_1, P_2, \ldots, P_m$) avec la même clé k, on peut calculer les $4P_i$ correspondant et sélectionner un bit quelconque s de leur représentation. Parmi les traces de mesures, on trie celles pour lesquelles le bit s serait à 0 (on pose alors $D(P_i, s) = 0$) ou à 1 (on pose alors $D(P_i, s) = 1$) pour faire les deux sous-ensembles.

On calcule ensuite la différentielle suivante :

$$\begin{aligned}\Delta_D[j] &= \frac{\sum_{i=1}^{m} D(P_i, s) \cdot \mathbf{T}_i[j]}{\sum_{i=1}^{m} D(P_i, s)} - \frac{\sum_{i=1}^{m} (1 - D(P_i, s)) \cdot \mathbf{T}_i[j]}{\sum_{i=1}^{m} (1 - D(P_i, s))} \\ &\approx 2 \cdot \left(\frac{\sum_{i=1}^{m} D(P_i, s) \cdot \mathbf{T}_i[j]}{\sum_{i=1}^{m} D(P_i, s)} - \frac{\sum_{i=1}^{m} \mathbf{T}_i[j]}{m} \right).\end{aligned}$$

On remarque que le cardinal de chaque sous-ensemble est proche de $m/2$ pour m suffisamment grand.

Lorsque l'hypothèse $k_{t-2} = 0$ est fausse, les valeurs de cette différentielle en tous points ne sont pas corrélées avec le calcul du bit s de $4P$. En effet, la sélection des traces pour la valeur supposée de k_{t-2} et le bit s de $4P$ résultant (correspondant au point de base connu de la mesure) diffère peu d'un point de vue statistique d'une sélection aléatoire dans ce cas. La moyenne des différences tend donc vers 0 : au bruit près, la différentielle Δ_D est plate.

En revanche, dans le cas contraire, lorsque l'hypothèse $k_{t-2} = 0$ s'avère juste, certains points de mesures correspondant au traitement de ce bit s de $4P$ vont dépendre de sa valeur, 0 ou 1. Pour ces points, la corrélation entre la puissance consommée par l'appareil et la valeur de ce bit fait apparaître une différence significative au niveau de Δ_D, correspondant à la différence de puissance consommée par l'appareil pour traiter un 0 ou un 1 pour le bit s. Des pics de valeurs sont alors visibles dans la zone considérée. On a alors déterminé le bit k_{t-2} de la clé utilisé pour la multiplication scalaire.

On voit que deux moyennes seulement sont à calculer par bit de clé. On procède ensuite de façon récursive pour retrouver l'ensemble des bits de la clé.

Coron dans [16] reproduit des mesures effectuées sur une multiplication scalaire de point de courbe elliptique effectuée par une carte à puce tout à fait convaincante sur l'efficacité de cette attaque pour la multiplication scalaire. Cette attaque s'adapte de la même manière au cas de l'exponentiation modulaire.

3.4.2 Principales contre-mesures

Trois parades sont mentionnées par Coron dans [16] comme contre-mesures dans le cas de l'attaque DPA. La première technique présente un certain intérêt aussi pour l'attaque SPA.

De par le traitement statistique qui est fait dans l'attaque DPA (le calcul de moyennes), il semble naturel d'envisager des parades qui rajoutent de l'aléatoire dans les opérations. La *randomization* des opérandes semble donc couler de source. Ces opérandes sont au nombre de trois : l'exposant (la clé privée k d'une multiplication scalaire d'un échange de clé de Diffie-Hellmann, par exemple), le point de base P, et enfin l'exploitation d'un changement de coordonnées pour la courbe elliptique (coordonnées projectives, par exemple).

On trouve une description de ces techniques par Coron dans [16].

3.4.2.1 Camouflage du scalaire k

Une courbe elliptique sur $\mathbb{F}_q$ comporte un nombre fini de points, que l'on note $\#E(\mathbb{F}_q)$, dont l'ordre divise $\#E(\mathbb{F}_q)$. L'idée est donc de multiplier le point de base par un scalaire qui est la somme du scalaire secret et d'un multiple de $\#E(\mathbb{F}_q)$. De par la propriété de groupe cyclique de l'ensemble des points de la courbe elliptique, le résultat est celui recherché. En équation, cela donne :

$$Q = k \cdot P = (k + \mathcal{R} \cdot \#E(\mathbb{F}_q)) \cdot P, \forall \mathcal{R} \in \mathbb{Z}.$$

(On a en effet : $\#E(\mathbb{F}_q) \cdot P = \mathcal{O}$).

En pratique, Coron dans [16] précise que $\mathcal{R}$ peut être pris comme un entier aléatoire d'une vingtaine de bits.

Ce que Coron ne précise pas, en revanche, c'est le surcoût de cette parade. On peut cependant l'estimer. En effet, il faut d'abord générer un nombre aléatoire (environ 20 bits, comme nous l'avons vu), ensuite, il faut multiplier ce nombre et $\#E(\mathbb{F}_q)$ (multiplication d'entier en multiprécision). L'exposant secret est normalement codé sur un nombre de bits au plus égal à celui de $\#E(\mathbb{F}_q)$, il est donc maintenant possible que l'exposant utilisé soit codé sur un nombre de bits au plus égal à celui de $\#E(\mathbb{F}_q) + 20$ bits. Ceci rajoute donc 20 tours de boucle à l'algorithme qui effectue l'ECSM. Il faudra donc en tenir compte, car cela représente autant de

doublements et moitié moins d'additions de points sur $\mathbb{F}_{2^m}$ ou $\mathbb{F}_q$ selon le cas, pour l'algorithme *Double-and-add* par exemple.

Dans le cas de l'attaque DPA, ces 20 bits supplémentaires offrent un brouillage des mesures et de la différentielle Δ_D suffisant, d'après Coron dans [16].

Une critique que formulent Ciet et Joye dans [14], c'est que dans le cas des courbes elliptiques préconisées par le NIST, le masquage est assez imparfait. En effet, l'ordre des points de la courbe $\#E(\mathbb{F}_q)$ est lui même creux. Pour la courbe B233 par exemple, on a
$\#E(\mathbb{F}_q) =$ `0x 00000100 00000000 00000000 00000000 0013E974 E72F8A69 22031D26 03CFE0D7`.
Multiplié par un facteur aléatoire sur 20 bits comme le recommande Coron, cela donnera un masque de la forme :
`0x 0XXXXX00 00000000 00000000 000000XX XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX`.
Le nombre de zéros dans le masque sont autant de bits du scalaire qui resteront visibles par l'attaque DPA. La fuite d'information reste donc non négligeable dans ce cas. Ceci concerne toutes les courbes elliptiques du NIST (voir [19]).

Une autre critique est formulée par Liardet et Smart dans [41]. Ils font remarquer que cette contre-mesure est inefficace si elle est appliquée à un algorithme non régulier vulnérable à l'attaque SPA. En effet le simple calcul de $(k + \mathcal{R} \cdot \#E(\mathbb{F}_q)) \bmod \#E(\mathbb{F}_q)$ est toujours possible pour l'attaquant si $\#E(\mathbb{F}_q)$ est connu, rendant inefficace cette parade. Ils décrivent donc une deuxième parade inspirée de cette première, qui consiste à générer un nombre aléatoire $\mathcal{R}$ et son inverse $\mathcal{R}' = \mathcal{R}^{-1} \bmod \#E(\mathbb{F}_q)$. À partir de là, on peut calculer alors :

$$\begin{array}{rcl} k' & = & \mathcal{R}' k \bmod \#\mathcal{E}, \\ \text{puis} \quad Q' & = & k' \cdot P \\ \text{et enfin} \quad Q & = & \mathcal{R} \cdot Q' = k \cdot P \end{array}$$

La parade ci-dessus présente un coût équivalent à la randomisation proposée par Coron pour $\mathcal{R}$ de taille 20 bits. Cette parade répond à la critique précédente sur le masquage creux. En revanche, si une attaque SPA donne la connaissance de k' et de $\mathcal{R}$, cela permet encore de retrouver k en calculant $k = k' \times \mathcal{R}' \bmod \#E(\mathbb{F}_q)$.

3.4.2.2 Camouflage du point P

Ici, il est proposé d'ajouter un point aléatoire R au point de base P dont on connaît le multiple à l'avance $S = k \cdot R$. Ces points peuvent être stockés par avance dans la mémoire de l'appareil pour éviter le surcoût d'un calcul. À chaque calcul, on peut également modifier R et S comme suit :

$$R \leftarrow (-1)^b 2R, \quad S \leftarrow (-1)^b 2S, \ \ (b \text{ un bit aléatoire}).$$

Ceci coûte deux doublements supplémentaires à chaque opération. Il est également fait remarquer que la connaissance de P ne donne pas d'avantage à l'attaquant, dans la mesure où il ne connaît pas le point utilisé dans l'opération $P' = P + R$.

3.4.2.3 Coordonnées projectives randomisées

Il y a pour un point en coordonnées affines (x, y) une grande quantité de représentations (X, Y, Z) en $\mathcal{LD}$ *projective*, telles que $(x = X/Z, y = Y/Z^2)$, en fait tous les points $(\lambda X, \lambda^2 Y, \lambda Z)$ (dans $\mathbb{F}_{2^m}$ ici, et $\lambda \in \mathbb{N}$). Ceci est valable pour les systèmes de coordonnées analogues (projectif, jacobien, $\mathcal{LD}$, *PL*...) et selon que nous avons une courbe sur $\mathbb{F}_{2^m}$ ou $\mathbb{F}_p$. Cette parade n'est pas gratuite :

— il faut générer une valeur λ aléatoire (ceci peut être fait à l'avance) ;

— il faut en premier lieu représenter le point de base qui est en coordonnées affines en un point en coordonnées projectives aléatoire à l'aide de λ, au coût de $2M + 1S$ pour les coordonnées $\mathcal{LD}$;
— l'algorithme de multiplication scalaire doit utiliser ce point de base en coordonnées projectives, et dans le cas de l'algorithme 2.4 *Left-to-right Double-and-add* (voir page 68), nous devons alors employer une addition en coordonnées projectives et non mixées dont la comparaison du coût est rappelée ici (voir la table 2.12 page 67) :

	coord.	opérations
Addition en coordonnées mixées	$\mathcal{LD}$	$9M + 4S + 13R$
Addition Projective	$\mathcal{LD}$	$13M + 4S + 17R$

En conclusion, le surcoût de cette parade est d'environ 25 % sur les additions de points.

Deuxième partie

Contrer l'attaque SPA plus efficacement

Chapitre 4

Combiner les opérations dans l'exponentiation modulaire

Nous appelons opérations combinées la réalisation conjointe de deux ou plusieurs opérations partageant un ou plusieurs opérandes communes en une seule fois. Pour illustrer ce concept, on peut donner un petit exemple avec une multiplication d'entiers. On veut multiplier 231 par 45, puis 327 par le même opérande 45. On peut effectuer les deux multiplications à la suite indépendamment l'une de l'autre. L'algorithme de l'écolier va fonctionner de la façon suivante pour chacune des opérations :

$$\begin{array}{rrrrrr} & & & 2 & 3 & 1 \\ \times & & & & 4 & 5 \\ \hline & & & & 4 & 5 \\ & & 1 & 3 & 5 & \\ & & 9 & 0 & & \\ \hline & 1 & 0 & 3 & 9 & 5 \end{array} \qquad \begin{array}{rrrrrr} & & & 3 & 2 & 7 \\ \times & & & & 4 & 5 \\ \hline & & & 3 & 1 & 5 \\ & & & 9 & 0 & \\ & 1 & 3 & 5 & & \\ \hline & 1 & 4 & 7 & 1 & 5 \end{array}$$

On remarque que les valeurs $2\times45 = 90$ et $3\times45 = 135$ sont calculées à chaque fois. Une idée pour économiser des calculs, c'est de vérifier les chiffres communs dans les opérande à multiplier par 45 et de n'effectuer que les multiplications partielles correspondantes. Dans notre exemple, on économise 2 multiplications partielles sur 6, soit 1/3 de la complexité sur les multiplications !

Nous appliquons ce concept d'opérations combinées dans le cas de l'exponentiation modulaire utilisant la multiplication de Montgomery. Nous tirons parti d'opérations avec un opérande commun AB, AC, puis $AB_1, \ldots, AB_\ell$. Ceci permet de mutualiser des calculs préalables pour toutes les multiplications concernées. Nous avons élaboré de nouveaux algorithmes qui ont fait l'objet d'implantation logicielle dans des exponentiations de type déchiffrement RSA basés sur l'échelle binaire de Montgomery et sur les exponentiations régulières β-aires. L'analyse de complexité montre que pour un module RSA de taille 2048 bits, les améliorations proposées réduisent le nombre d'opérations élémentaires sur mots machine (ADD et MUL) de 14% pour l'échelle binaire de Montgomery et 5%–8% pour les exponentiations régulières β-aires. Notre implantation logicielle présente une accélération de 8%–14% pour l'échelle binaire de Montgomery et 1%–8% pour les exponentiations β-aires, pour des modules de tailles 1024, 2048 et 4096 bits.

Ce travail a été présenté lors de la conférence Arith 22 à Lyon le 24 juin 2015, et a été publié dans les actes de cette conférence [50].

4.1 Multiplications de Montgomery multiples partageant un opérande commun

Nous avons présenté la multiplication modulaire de Montgomery au chapitre 1, section 1.1.3, pages 24 et suivantes. Les algorithmes d'exponentiation consistent en une séquence de multiplications et d'élévations au carré. Dans cette séquence, certains opérandes peuvent se voir utilisés plusieurs fois. Notre objectif est donc de tirer parti de cette réutilisation, en mutualisant des calculs communs aux multiplications partageant un même opérande.

On suppose ici et dans la suite que les opérandes et le module sont représentés par n mots de w bits. Rappelons la boucle principale de l'algorithme 1.8 (page 30) :

for $i = 1$ **to** $n - 1$ **do**
  $Y \leftarrow Y + a_i \cdot B$
  $q \leftarrow |Y|_{2^w} \cdot N' \bmod 2^w$
  $Y \leftarrow (Y + q \cdot N)/2^w$
end for
return Y

Cette boucle principale consiste en une accumulation et une petite réduction *smallRed* (voir l'algorithme 1.7 page 30). Lorsque l'on a plusieurs multiplications de ce type à effectuer avec un opérande commun, nous proposons de mutualiser les *smallReds* par une réécriture de la multiplication de Montgomery.

Dans cette section, nous traitons en premier le cas où l'on doit effectuer deux multiplications $A \cdot B, A \cdot C$, puis nous généraliserons ce résultat au cas de ℓ multiplications $A \cdot B_i, i = 1, \ldots, \ell$.

4.1.1 Deux multiplications combinées de type $A \cdot B, A \cdot C$

Nous présentons dans cette section notre nouvel algorithme pour deux multiplications combinées de type $A \cdot B, A \cdot C$. Ensuite, nous donnons la preuve de sa validité.

Deux multiplications partagent un même opérande. Les entrées de ces calculs seront les trois opérandes A, B, C et le module N, et les sorties seront $ABR^{-1} \bmod N$ et $ACR^{-1} \bmod N$ avec $R = 2^{w(n+1)}$, la constante de Montgomery. Le lecteur peut d'ores et déjà remarquer que nous l'avons modifiée par rapport à la multiplication de Montgomery classique, qui définit cette constante comme $M = 2^{wn}$.

Notre objectif est de mutualiser les calculs communs effectués dans les produits $ABR^{-1} \bmod N$ et $A\,CR^{-1} \bmod N$. Cet objectif est réalisé en récrivant la multiplication ABR^{-1} en développant en fonction de $B = \sum_{j=0}^{n-1} b_j 2^{wj}$ de la façon suivante :

$$\begin{aligned}
ABR^{-1} &= \left(\sum_{j=0}^{n-1} b_j 2^{wj}\right) \cdot A \cdot 2^{-w(n+1)} \bmod N \\
&= \sum_{j=0}^{n-1} b_j A \cdot 2^{-w(n+1-j)} \bmod N \\
&= \sum_{j=0}^{n-1} b_j \left(A \cdot 2^{-w(n-1-j)} \bmod N\right) 2^{-2w} \bmod N \\
&= \left(\sum_{j=0}^{n-1} b_j A^{(j)}\right) \cdot 2^{-2w} \bmod N \qquad (4.1)
\end{aligned}$$

où $A^{(j)} = A2^{-w(n-1-j)} \bmod N$ pour $j = 0, \ldots, n-1$. On peut faire de même avec $A \cdot C \cdot R^{-1}$, et on obtient alors

$$A \cdot C \cdot R^{-1} = \left(\sum_{j=0}^{n-1} c_j A^{(j)} \right) \cdot 2^{-2w} \bmod N. \tag{4.2}$$

On remarque que dans l'expression (4.1) correspondant à ABR^{-1} et dans l'expression (4.2) correspondant à ACR^{-1}, les mêmes termes $A^{(j)} = A2^{-w(n-1-j)} \bmod N, j = 0, 1, \ldots, n-1$ sont présents. Ainsi, nous proposons de ne calculer ces termes qu'une seule fois pour les deux multiplications.

Ces termes $A^{(j)}$ se déduisent de $A^{(n-1)} = A$, et les suivants sont obtenus en appliquant $n-1$ petites réductions de Montgomery d'un mot à la fois (*smallRed*(X), voir l'algorithme 1.7 page 30). On rappelle que pour cet algorithme, nous avons en entrée $X < 2^w \cdot M$ et en sortie nous calculons $(X = X2^{-w} \bmod N)$. Voici donc la séquence de calcul des $A^{(j)}$:

$$\begin{aligned}
\textit{smallRed}(A^{(n-1)}) &= A \cdot 2^{-w} \bmod N \\
&= A^{(n-2)}, \\
\textit{smallRed}(A^{(n-2)}) &= (A \cdot 2^{-w}) \cdot 2^{-w} \bmod N \\
&= A^{(n-3)}, \\
&\vdots \qquad\qquad \vdots \\
\textit{smallRed}(A^{(1)}) &= \left(A \cdot 2^{-w(n-2)}\right) \cdot 2^{-w} \bmod N \\
&= A \cdot 2^{-w(n-1)} = A^{(0)}.
\end{aligned}$$

Maintenant, on peut calculer AB et AC en accumulant les résultats des multiplications $b_j A^{(j)}$ et $c_j A^{(j)}$ dans Y et Z pour $j = 0, \ldots, n-1$. Le lemme 1.1.1 page 29 indique que les valeurs $A^{(j)}, j = 0, \ldots, n-1$, satisfont $A^{(j)} < 2N$ si A est inférieur à $2N$. Par conséquent, l'accumulation $\sum_{j=0}^{n-1} b_j A^{(j)} < 2Nn2^w$ (dans Y) est réduite à son tour en valeur inférieure à $2N$ en lui appliquant deux *smallReds* successives. Ces deux dernières opérations produisent le facteur 2^{-2w} dans l'équation (4.1). Les mêmes opérations sont effectuées pour le calcul de Z.

L'algorithme 4.1 matérialise cette démarche, et sa complexité est evaluée pas-à-pas dans la table 4.1. Dans cette table et dans la suite, nous notons ADD une addition élémentaire de deux opérandes de taille w bits, ou encore un mot machine de taille w bits. De façon similaire, on note MUL une multiplication élémentaire de deux opérandes de taille w bits, ou encore un mot machine de taille w bits. Dans ce cernier cas, le résultat est de taille $2w$ bits. Ces complexités se déduisent des résultats de complexité des algorithmes 1.3 page 25 (multiplication multi-précision $1 \times n$) et 1.7 page 30 (*smallRed*) et fournies dans les tables 1.4 page 28 et 1.7 page 33.

4.1.2 Multiplications multiples partageant un opérande commun

Dans cette section, nous présentons l'extension de l'idée précédente au cas de multiplications multiples partageant un opérande commun. En pratique, étant donnés $A \in \{0, \ldots, N-1\}$ et une série de valeurs $B_i, i = 1, \ldots, \ell$, on souhaite obtenir :

$$\begin{aligned}
Y_1 &= A \cdot B_1 \cdot 2^{-w(n+1)} \bmod N, \\
Y_2 &= A \cdot B_2 \cdot 2^{-w(n+1)} \bmod N, \\
&\vdots \\
Y_\ell &= A \cdot B_\ell \cdot 2^{-w(n+1)} \bmod N.
\end{aligned}$$

Algorithme 4.1 *CombinedMontMul*(A, B, C)

Require: $N < 2^{wn-1}$ le module, $A = (a_{n-1}, \dots a_0)_2, B = (b_{n-1}, \dots b_0)_2, C = (c_{n-1}, \dots c_0)_2$ trois entiers tels que $A, B, C < 2N$, w la taille du mot machine, la constante de Montgomery $R = 2^{w(n+1)}$.

Ensure: $Y = A \cdot B \cdot R^{-1} \bmod N$ and $Z = A \cdot C \cdot R^{-1} \bmod N$

1: $X \leftarrow A$
2: $Y \leftarrow b_{n-1} \cdot X,\ Z \leftarrow c_{n-1} \cdot X$
3: **for** $j = n - 2$ **downto** 0 **do**
4: $\quad q \leftarrow |X|_{2^w} N' \bmod 2^w$
5: $\quad X \leftarrow (X + q \cdot N)/2^w \quad // = A^{(j)}$ for $j = n-2, \dots, 0$
6: $\quad Y \leftarrow Y + b_j \cdot X,\ Z \leftarrow Z + c_j \cdot X$
7: **end for**
8: $Y \leftarrow$*smallRed*$(Y),\ Z \leftarrow$*smallRed*(Z)
9: $Y \leftarrow$*smallRed*$(Y),\ Z \leftarrow$*smallRed*(Z)
10: **return** Y and Z

TABLE 4.1 – Complexité de l'algorithme 4.1 *CombinedMontMul*

	Opérations	# ADD	# MUL
Étape 2	$b_{n-1} \cdot X$	n	n
Étape 2	$c_{n-1} \cdot X$	n	n
$(n-1)$ Étape 4	$\lvert X\rvert_{2^w} \cdot N'$	0	$n-1$
$(n-1)$ Étape 5	$q \cdot N$ $X + (qN)$	$(n-1)n$ $(n-1)(n+1)$	$(n-1)n$ 0
$(n-1)$ Étape 6	$b_j \cdot X$ $Y + (b_j X)$	$(n-1)n$ $(n-1)(n+2)$	$(n-1)n$ 0
$(n-1)$ Étape 6	$c_j \cdot X$ $Z + (c_j X)$	$(n-1)n$ $(n-1)(n+2)$	$(n-1)n$ 0
Étape 8	$2\times$*smallRed* avec $n' = n+2$	$4n+4$	$2n+2$
Étape 9	$2\times$ *smallRed* avec $n' = n+1$	$4n+2$	$2n+2$
Total		$6n^2+9n+1$	$3n^2+4n+3$

Comme en section 4.1.1, nous récrivons chaque multiplication $A \cdot B_i \cdot 2^{-w(n+1)} \bmod N$ en développant en fonction de $B_i = \sum_{j=0}^{n-1} b_{i,j} 2^{wj}$ comme suit :

$$\begin{aligned} A \cdot B_i \cdot 2^{-w(n+1)} &= (\sum_{j=0}^{n-1} b_{i,j} 2^{wj}) \cdot A^{-w(n+1)} \bmod N \\ &= (\sum_{j=0}^{n-1} b_{i,j} A^{(j)}) \cdot 2^{-2w} \bmod N. \end{aligned} \tag{4.3}$$

Les expressions $A \cdot B_i \cdot 2^{-w(n+1)}$ contiennent les mêmes valeurs $A^{(j)} = A \cdot 2^{-w(n-1-j)} \bmod N$ pour $j = 0, \ldots, n-1$. Notre idée consiste donc à précalculer ces valeurs et à les stocker en mémoire. Elles s'obtiennent aisément par une suite de *smallReds* comme on peut le voir dans l'algorithme 4.2.

Algorithme 4.2 *PrecompMultByComOp*(A)

Require: $A, N \in \mathbb{Z}$ avec $A < 2N$ et $N < 2^{nw-2}$, $N' = (-N^{-1}) \bmod 2^w$ précalculé.
Ensure: $A^{(0)}, \ldots, A^{(n-1)}$ tels que $A^{(j)} = 2^{-w(n-1-j)} \cdot A \bmod N$ et $A^{(j)} < 2N$
1: $A^{(n-1)} \leftarrow A \bmod N$
2: **for** $j = n-2$ **downto** 0 **do**
3: $\quad A^{(j)} \leftarrow$*smallRed*$(A^{(j+1)})$
4: **end for**
5: **return** $A^{(0)}, \ldots, A^{(n-1)}$

Complexité de l'algorithme 4.2. Lors de cet algorithme, on calcule $n-1$ *smallReds*, toutes avec une valeur de taille n-mots en entrée. Connaissant la complexité de chacune des *smallReds* (voir la table 1.7 page 33), on obtient celle de l'algorithme 4.2, qui est $(2n+1)(n-1)$ ADD et (n^2-1) MUL.

Maintenant, avec les valeurs précalculées $A^{(0)}, \ldots, A^{(n-1)}$, il est possible de calculer chacun des produits $A \cdot B_i \cdot 2^{-w(n+1)} \bmod N$ avec $i = 1, \ldots, \ell$ en utilisant l'équation (4.3). Cette stratégie est utilisée pour calculer $A \cdot B \cdot 2^{-(w(n+1)} \bmod N$ quelle que soit l'entrée B appartenant à $[0, 2N]$ dans l'algorithme 4.3.

Algorithme 4.3 *MultByComOp*($B, A^{(0)}, \ldots, A^{(n-1)}$)

Require: Le module $N < 2^{wn-1}$, un entier $B = (b_{n-1}, \ldots b_0)_{2^w}$ tel que $B < 2N$ et $A^{(j)} = 2^{-w(n-1-j)} \cdot A \bmod N$ pour $j = 0, \ldots, n-1$.
Ensure: $Y = A \cdot B \cdot 2^{-w(n+1)} \bmod N$ avec $Y < 2N$
1: $Y \leftarrow b_0 \cdot A^{(0)}$
2: **for** $j = 1$ **to** $n-1$ **do**
3: $\quad Y \leftarrow Y + b_j \cdot A^{(j)}$
4: **end for**
5: $Y \leftarrow$*smallRed*(Y)
6: $Y \leftarrow$*smallRed*(Y)
7: **return** Y

La table 4.2 fournit la complexité de l'algorithme 4.3 évaluée pas à pas.

Le calcul de ℓ multiplications $A \cdot B_i \cdot 2^{-w(n+1)} \bmod N$ pour $i = 1, \ldots, \ell$ consiste en une exécution de l'algorithme 4.2 et ℓ exécutions de l'algorithme 4.3. ℓ multiplications calculées à l'aide de cette stratégie coûtent donc $\ell(2n^2+5n+1) + (2n+1)(n-1)$ ADD et $\ell(n^2+2n+2) + (n^2-1)$ MUL.

TABLE 4.2 – Complexité de l'algorithme 4.3 *MultByComOp*

	Opérations	# ADD	# MUL
Étape 1	$b_0 \cdot A^{(0)}$	n	n
Étape 3	$b_j \cdot A^{(j)}$	$(n-1)n$	$(n-1)n$
$\times(n-1)$	$Y + (b_j A^{(j)})$	$(n-1)(n+2)$	0
Étape 5	*smallRed* avec $n' = n+2$	$2n+2$	$n+1$
Étape 6	*smallRed* avec $n' = n+1$	$2n+1$	$n+1$
Total		$2n^2+5n+1$	n^2+2n+2

4.1.3 Comparaison des différentes complexités

Nous comparons les complexités des algorithmes 4.1 (*CombinedMontMul*) et 4.3 (*MultByComOp*) avec celles des approches courantes qui effectuent deux multiplications modulaires de Montgomery (l'algorithme 1.8 page 1.8) indépendantes pour AB, AC, mais aussi une multiplication modulaire et une élévation au carré de Montgomery (l'algorithme 1.9 page 1.9) pour les opérations de type AB, A^2. Enfin pour ℓ multiplications par un opérande commun, nous comparons avec le calcul de ℓ *MontMuls*.

TABLE 4.3 – Comparaison des complexités pour le calcul de AB, AC et AB, A^2.

Operation	Algorithme	# ADD	# MUL
AB, AC	$2\times$ *MontMul*s Algo. 1.8	$8n^2+4n-2$	$4n^2+2n$
AB, A^2	*MontMul* Algo. 1.8 + *MontSq* Algo. 1.9	$7n^2+7n-2$	$\frac{7}{2}n^2+\frac{7}{2}n-1$
AB, AC	*CombinedMontMul* Algo. 4.1	$6n^2+9n+1$	$3n^2+4n+3$
AB_i, $i=1,\ldots,\ell$	ℓ *MontMul*s Algo. 1.8	$\ell(4n^2+2n-1)$	$\ell(2n^2+n)$
AB_i, $i=1,\ldots,\ell$	$\ell\times$ *MultByComOp* Algo. 4.3 $1\times$ PrecomMultByComOp Algo. 4.2	$\ell(2n^2+5n+1)$ $+(2n+1)(n-1)$	$\ell(n^2+2n+2)$ $+(n^2-1)$

Le lecteur peut vérifier que

— notre opération *CombinedMontMul* pour le calcul de AB, AC apporte une diminution du nombre d'opérations de l'ordre de 25 % par rapport au calcul de deux multiplications *MontMul*s indépendantes.

— notre opération *CombinedMontMul* pour le calcul de AB, A^2 apporte une diminution du nombre d'opérations de l'ordre de 13 % par rapport au calcul d'une multiplication *MontMul*s suivie d'un carré *MontSq*.

— notre opération *MultByComOp*, effectuant ℓ multiplications $AB_1, \ldots, AB_\ell$, apporte également une amélioration de 25% en comparaison de ℓ multiplications indépendantes *MontMul*s, et jusqu'à environ 50% lorsque ℓ devient grand, rendant le coût des précalculs moins significatif.

4.2 Exponentiation modulaire avec multiplications multiples partageant un opérande commun

Après avoir élaboré de nouvelles opérations multiples tirant parti d'un opérande commun, nous présentons maintenant leur application aux algorithmes d'exponentiation régulière vus au chapitre précédent : l'échelle binaire de Montgomery (l'algorithme 3.5 page 84) ainsi que les exponentiations régulières proposées par Joye et Tunstall dans [31] que nous avons présentées dans les algorithmes 3.9 et 3.10 page 88. Notre objectif ici est de conserver le comportement régulier gage de résistance à l'attaque SPA, tout en améliorant leurs performances.

4.2.1 Échelle binaire de Montgomery avec *CombinedMontMul*

L'échelle binaire de Montgomery évalue l'exponentiation modulaire $G^e \bmod N$ par le moyen d'opérations de la forme $A \cdot B, A \cdot A$. Notre opération *CombinedMontMul* peut être utilisée dans ce cas de la manière suivante : *CombinedMontMul*(A, B, A) renvoie $A \cdot B$ et $A \cdot A$.

Reprenons les complexités de la table 4.3. Nous remarquons que l'utilisation de *CombinedMontMul* est moins coûteuse que le calcul des opérations *MontMul* et *MontSq*. La complexité s'en trouve donc réduite d'autant. L'adaptation de l'échelle binaire de Montgomery à l'utilisation de notre opération *CombinedMontMul* nécessite toutefois l'ajout de quelques *smallReds* pour tenir compte de la nouvelle constante de Montgomery dans les multiplications modulaires. Ceci est montré dans l'algorithme 4.4.

Algorithme 4.4 Montgomery-ladder avec *CombinedMontMul*

Require: $N < 2^{wn-1}$ et $g \in \{0, \ldots, N-1\}$, un exposant $e = (e_{t-1}, \ldots e_0)_2$, la taille d'un mot machine w et la constante de Montgomery $R = 2^{w(n+1)}$.
Ensure: $g^e \bmod N$
1: $X_0 \leftarrow R \bmod N$
   //conversion $X_1 \leftarrow g \cdot R^2 \cdot R^{-1} \bmod N$
2: $X_1 \leftarrow$ *MontMul*$(g, R^2 \bmod N)$
3: $X_1 \leftarrow$ *smallRed*(X_1)
4: **for** $i = t-1$ **downto** 0 **do**
5: $\quad X_{1-e_i}, X_{e_i} \leftarrow$ *CombinedMontMul*$(X_{e_i}, X_{1-e_i}, X_{e_i})$
6: **end for**
   // conversion $X_0 \leftarrow (g^e \cdot R) \cdot R^{-1} \bmod N$
7: $X_0 \leftarrow$ *MontMul*$(X_0, 1), X_0 \leftarrow$ *smallRed*(X_0)
8: **return** X_0

Complexité de l'algorithme 4.4. Les opérations effectuées par l'algorithme 4.4 sont :
— deux multiplications de Montgomery ;
— deux *smallReds* ;
— t *CombinedMontMul*s (correspondant aux t tours de la boucle **for** étape 4).
En reprenant les complexités de la table 4.3, on obtient :

$$\begin{aligned} \#\text{ADD} &= t(6n^2 + 9n + 1) + (8n^2 + 8n), \\ \#\text{MUL} &= t(3n^2 + 4n + 3) + (4n^2 + 4n + 2). \end{aligned}$$

4.2.2 *Regular right-to-left* 2^γ*-ary Exponentiation* avec *CombinedMontMul*

Nous avons présenté l'algorithme *Regular right-to-left* 2^γ*-ary Exponentiation* pour le calcul de $g^e \bmod N$ lors de notre revue des contre-mesures à l'attaque SPA (l'algorithme 3.10 page

88). Nous supposerons dans la suite que l'exposant e est recodé selon la méthode présentée par Joye et Tunstall en [31] (voir l'algorithme 3.8 page 87).

On a donc $e = (e_{\ell-1}, \ldots, e_0)_\beta$ où $e_i \in \{1, \ldots, \beta\}$ et $\beta = 2^\gamma$. La version *Regular right-to-left* 2^γ*-ary Exponentiation* s'appuie sur l'écriture de $g^e \bmod N$ suivante :

$$\begin{aligned} g^e \bmod N &= \prod_{i=0,e_i=1}^{i=\ell-1} g^{\beta^i} \cdot \prod_{i=0,e_i=2}^{i=\ell-1} g^{2\cdot\beta^i} \cdots \prod_{i=0,e_i=\beta}^{i=\ell-1} g^{\beta\cdot\beta^i} \bmod N \\ &= \prod_{j=1}^{\beta} Y_j^j \bmod N \end{aligned}$$

avec $Y_j = \prod_{i=0,e_i=j}^{i=\ell-1} g^{\beta^i} \bmod N$.

1. Suivant cette expression, $g^e \bmod N$ est calculé par le moyen d'une suite $X_i \leftarrow g^{\beta^i} \bmod N$ suivie d'une accumulation $Y_{e_i} \leftarrow Y_{e_i} \cdot X_i$ pour $i = 0, \ldots, \ell - 1$. À l'issue de cette accumulation, on obtient les valeurs Y_j pour $j = 1, \ldots, \beta$. Quand $\beta = 2^\gamma$, on peut tirer parti de notre opération *CombinedMontMul* pour améliorer l'efficacité de la méthode *Regular right-to-left* 2^γ*-ary Exponentiation*. On constate en effet que les opérations $Y_{e_i} \leftarrow Y_{e_i} \cdot X_i$ et $X_{i+1} \leftarrow X_i^{2^\gamma}$ peuvent être effectuées à l'aide de *CombinedMontMul* comme suit :
   $Y_{e_i}, X \leftarrow$ *CombinedMontMul*(X, Y_{e_i}, X)
   for $j = 1$ **to** $\gamma - 1$ **do**
   $X \leftarrow$ *MontSq*$(X), X \leftarrow$ *smallRed*(X)
   end for
2. Le résultat final $g^e = \prod_{j=1}^{\beta} Y_j^j$ est finalement évalué par $2(\beta-1)$ multiplications :
   $Z \leftarrow Y_\beta$
   for $i = \beta - 1$ **downto** 1 **do**
   $Y_i \leftarrow Y_i \cdot Y_{i+1} \bmod N$
   $Z \leftarrow Z \cdot Y_i \bmod N$
   end for

   On peut vérifier qu'à la fin, $Z = g^e \bmod N$. En conséquence, la reconstruction finale peut également être améliorée par l'utilisation de *CombinedMontMul*. On évalue en effet $Z = \prod_{j=1}^{\beta} Y_j^j$, soit deux multiplications avec un opérande commun $Y_i \cdot Y_{i+1} \bmod N$ et $Z \cdot Y_i \bmod N$, et ceci est répété $2^\gamma - 2$ fois. Nous pouvons donc remplacer ces opérations par $2^\gamma - 2$ *CombinedMontMul*s.

L'algorithme résultant est le *Regular right-to-left* 2^γ*-ary Exponentiation* montré dans l'algorithme 4.5. Nous évaluons sa complexité pas à pas dans la table 4.4.

4.2.3 *Regular left-to-right* 2^γ*-ary Exponentiation* avec *MultByComOp*

Nous considérons maintenant le petit frère de l'algorithme précédent, le *Regular left-to-right* 2^γ*-ary exponentiation*. Nous utiliserons cette fois notre opération de multiplications multiples partageant un opérande commun *MultByComOp*, que nous avons présentée section 4.1.2 page 101. Nous supposerons là encore dans la suite que l'exposant e est recodé selon la méthode présentée par Joye et Tunstall en [31]. On rappelle que l'exponentiation régulière *left-to-right* effectue en premier lieu les pré-calculs $Y_i = g^i \bmod N, i = 1, \ldots, 2^\gamma$ et ensuite évalue $X = g^e \bmod N$ par une suite de multiplications $X \leftarrow X^{2^\gamma} \cdot Y_{e_i}$ pour $i = \ell - 1, \ldots, 0$.

Nous tirons parti de notre opération *MultByComOp* de la façon suivante :

— Au commencement, on effectue les calculs préliminaires de $Y^{(0)}, \ldots, Y^{(n-1)}$ avec l'algorithme PrecomMultByComOp(Y_1) avec $Y_1 = g \cdot 2^{w(n+1)} \bmod N$ (l'algorithme 4.2 page 103), ceci dans le but de calculer efficacement $2^\gamma - 1$ multiplications $Y_i \leftarrow Y_1 \cdot Y_{i-1} \bmod N$.

Algorithme 4.5 *Regular right-to-left 2^γ-ary Exponentiation* avec *CombinedMontMul*

Require: le module $N < 2^{wn-1}$, $0 \leq g < N$ entier, $e = (e_{\ell-1}, \ldots, e_0)_{2^\gamma}$ l'exposant avec $e_i \in \{1, \ldots, 2^\gamma\}$, la constante de Montgomery $R = 2^{w(n+1)}$.
Ensure: $g^e \bmod N$

// $X = g \cdot R \bmod N$
1: $X \leftarrow$ *MontMul*$(g, R^2 \bmod N), X \leftarrow$ *smallRed*(X)
2: **for** $i = 1$ to 2^γ **do**
3: $\quad Y_i \leftarrow R \bmod N$
4: **end for**
5: **for** $i = 0$ to $\ell - 1$ **do**
6: $\quad Y_{e_i}, X \leftarrow$ *CombinedMontMul*(X, Y_{e_i}, X)
7: $\quad$ **for** $j = 1$ **to** $\gamma - 1$ **do**
8: $\quad\quad X \leftarrow$ *MontSq*$(X), X \leftarrow$ *smallRed*(X)
9: $\quad$ **end for**
10: **end for**
// Reconstruction finale
11: $Z \leftarrow Y_{2^\gamma}$
12: $Y_{2^\gamma-1} \leftarrow$ *MontMul*$(Y_{2^\gamma-1}, Y_{2^\gamma})$
13: $Y_{2^\gamma-1} \leftarrow$ *smallRed*$(Y_{2^\gamma-1})$
14: **for** $i = 2^\gamma - 1$ **downto** 2 **do**
15: $\quad Z, Y_{i-1} \leftarrow$ *CombinedMontMul*(Y_i, Z, Y_{i-1})
16: **end for**
17: $Z \leftarrow$ *MontMul*(Z, Y_1), $Z \leftarrow$ *smallRed*(Z)
18: $Z \leftarrow$ *MontMul*$(Z, 1), Z \leftarrow$ *smallRed*(Z)
19: **return** Z

TABLE 4.4 – Complexité de l'algorithme 4.5 *Regular right-to-left 2^γ-ary Exponentiation* avec *CombinedMontMul*

	Op.	# ADD	# MUL
Étape 1	MM	$4n^2 + 2n - 1$	$2n^2 + n$
	SR	$2n + 1$	$n + 1$
ℓ Étape 6	CMM	$\ell(6n^2 + 9n + 1)$	$\ell(3n^2 + 4n + 3)$
$(\gamma - 1)\ell$	MS	$(\gamma - 1)\ell(3n^2 + 5n - 1)$	$(\gamma - 1)\ell(\frac{3n^2}{2} + \frac{5n}{2} - 1)$
Étape 8	SR	$(\gamma - 1)\ell(2n + 1)$	$(\gamma - 1)\ell(n + 1)$
Étape 12	MM	$4n^2 + 2n - 1$	$2n^2 + n$
Étape 13	SR	$2n + 1$	$n + 1$
$(2^\gamma - 2)$	CMM	$(2^\gamma - 2)(6n^2 + 9n + 1)$	$(2^\gamma - 2)(3n^2 + 4n + 3)$
Étape 15			
Étape 17	MM	$4n^2 + 2n - 1$	$2n^2 + n$
	SR	$2n + 1$	$n + 1$
Étape 18	MM	$4n^2 + 2n - 1$	$2n^2 + n$
	SR	$2n + 1$	$n + 1$
Total		$\gamma\ell(3n^2 + 7n) + \ell(3n^2 + 2n + 1)$ $+2^\gamma(6n^2 + 9n + 1) + 4n^2 - 2n - 2$	$\gamma\ell(\frac{3n^2}{2} + \frac{7n}{2}) + \ell(\frac{3n^2}{2} + \frac{n}{2} + 3)$ $+2^\gamma(3n^2 + 4n + 3) + 2n^2 - 2$

CMM=*CombinedMontMul*, MM=*MontMul*, MS=*MontSq*, SR=*smallRed*

— Ensuite, pour chacun des Y_i, on calcule $Y_i^{(j)} = Y_i \cdot 2^{-w(n-1-j)} \bmod N$ pour $j = 0, \ldots, n-1$ en effectuant de même *PrecompMultByComOp*(Y_i).
— Dans la boucle principale, on calcule séquentiellement $X \leftarrow X^{2^\gamma} \cdot Y_{e_i}$ pour $i = \ell - 1, \ldots, 0$, et chacune des multiplications Y_{e_i} est effectuée par

$$X \leftarrow \textit{MultByComOp}(X, Y_{e_i}^{(0)}, \ldots, Y_{e_i}^{(n-1)}).$$

Ceci est transcrit dans l'algorithme 4.6, qui est donc la version améliorée de l'exponentiation régulière *left-to-right*. On remarquera que la suite des carrés X^{2^γ} nécessite dans ce cas autant de *smallReds*, afin de conserver le facteur de Montgomery correct, à savoir $R = 2^{-w(n+1)}$, et ainsi préserver la stabilité de la représentation de Montgomery.

Algorithme 4.6 *Regular left-to-right 2^γ-ary Exponentiation* avec *MultByComOp*

Require: Le module $N < 2^{wn-1}$, $0 \leq g < N$ entier, l'exposant $e = (e_{\ell-1}, \ldots, e_0)_{2^\gamma}$ avec $e_i \in \{1, \ldots, m\}$, La constante de Montgomery $R = 2^{w(n+1)}$.
Ensure: $g^e \bmod N$
// $Y_1 = g \cdot 2^{w(n+1)} \bmod N$
$Y_1 \leftarrow \textit{MontMul}(g, R^2 \bmod N)$
$Y_1 \leftarrow \textit{smallRed}(Y_1)$
$X \leftarrow R$
$Y_1^{(0)}, \ldots, Y_1^{(n-1)} \leftarrow \textit{PrecompMultByComOp}(Y_1)$
for $i = 2$ **to** 2^γ **do**
  $Y_i \leftarrow \textit{MultByComOp}(Y_{i-1}, Y_1^{(0)}, \ldots, Y_1^{(n-1)})$
  $Y_i^{(0)}, \ldots, Y_i^{(n-1)} \leftarrow \textit{PrecompMultByComOp}(Y_i)$
end for
for $i = \ell - 1$ **downto** 0 **do**
  for $j = 1$ **to** γ **do**
    $X \leftarrow \textit{MontSq}(X), X \leftarrow \textit{smallRed}(X)$// $X \leftarrow X^2 \cdot 2^{-w(n+1)} \bmod N$
  end for
  $X \leftarrow \textit{MultByComOp}(X, Y_{e_i}^{(0)}, \ldots, Y_{e_i}^{(n-1)})$
end for
// $X \leftarrow X \cdot 2^{-w(n+1)} \bmod N$
$X \leftarrow \textit{MontMul}(X, 1), X \leftarrow \textit{smallRed}(X)$
return X

Remarque 4.2.1. *Nous soulignons ici que notre version de* Regular left-to-right 2^γ-ary Exponentiation *nécessite de stocker un grand nombre de valeurs, en fonction de la taille de la fenêtre γ. La taille totale de la mémoire mobilisée est*

$$\cong n \times 2^\gamma \times nw \textit{ bits, pour le stockage de } n \times 2^\gamma \textit{ termes } Y_i^{(j)}.$$

4.2.4 Comparaisons des complexités

Dans la table 4.6, nous donnons les complexités correspondantes pour les trois algorithmes que nous proposons : l'échelle binaire de Montgomery, *Regular right-to-left* et *left-to-right 2^γ-ary Exponentiations*.

Pour chacune de ces méthodes d'exponentiations, nous donnons premièrement les complexités sans aucune optimisation, c'est-à-dire celles de l'état de l'art, en utilisant les multiplications et carrés de Montgomery courants. Nous donnons ensuite les complexités de nos versions améliorées, c'est-à-dire les algorithmes 4.4, 4.5 et 4.6. Quelques commentaires sur cette table vont aider dans l'interprétation des résultats.

TABLE 4.5 – Complexité de l'algorithme 4.6 *Regular left-to-right 2^γ-ary Exponentiation* avec *MultByComOp*

	Op.	# ADD	# MUL
Étape 1	MM	$4n^2+2n-1$	$2n^2+n$
Étape 2	SR	$2n+1$	$n+1$
Étape 4	PMBCO	$2n^2-n-1$	n^2-1
$(2^\gamma-1)$ Étape 6	MBCO	$(2^\gamma-1)(2n^2+5n+1)$	$(2^\gamma-1)(n^2+2n+2)$
$(2^\gamma-1)$ Étape 7	PMBCO	$(2^\gamma-1)(2n^2-n-1)$	$(2^\gamma-1)(n^2-1)$
$\gamma\ell$ Étape 11	MS SR	$\gamma\ell(3n^2+5n-1)$ $\gamma\ell(2n+1)$	$\gamma\ell(\frac{3n^2}{2}+\frac{5n}{2}-1)$ $\gamma\ell(n+1)$
ℓ Étape 13	MBCO	$\ell(2n^2+5n+1)$	$\ell(n^2+2n+2)$
Étape 15	MM SR	$4n^2+2n-1$ $2n+1$	$2n^2+n$ $n+1$
Total		$\gamma\ell(3n^2+7n)+\ell(2n^2+5n+1)$ $+2^\gamma(4n^2+4n)+6n^2+3n-1$	$\gamma\ell(\frac{3n^2}{2}+\frac{7n}{2})+\ell(n^2+2n+2)$ $+2^\gamma(2n^2+2n+1)+3n^2+2n$

PMBCO=PrecompMulByComOp, MBCO=MulByComOp,
MM=MontMul, MS=*MontSq*, SR=*smallRed*

Nous donnons également en référence la complexité de l'algorithme *Square-always* proposé par Clavier *et al.* dans [15], que nous avons présenté en section 3.2.2.2 page 82. Nous n'avons en revanche pas donné celles des approches *Square-and-multiply-always* de Coron dans [16] et *Square-multiply* de Joye dans [30], considérant que leur complexité est identique à celle de l'échelle binaire de Montgomery.

Premièrement, considérons l'échelle binaire de Montgomery. L'amélioration procurée par notre opération *CombinedMontMul* diminue le terme le plus significatif pour le nombre d'opération ADD de $7tn^2$ à $6tn^2$. De même, pour le terme le plus significatif pour le nombre d'opération MUL, on observe une diminution de $3,5tn^2$ à $3tn^2$. Ceci représente une diminution de 14% par rapport à l'échelle binaire de Montgomery, et rend notre approche équivalente à l'algorithme *Square-always* de Clavier *et al.* dans [15], qui est régulier mais dont le temps global d'exécution n'est pas constant.

Deuxièmement, dans le cas des méthodes dites *Regular right-to-left 2^γ-ary Exponentiation*, les termes les plus significatifs sont ceux en $\gamma\ell$. Ils sont globalement équivalents dans les deux cas, version standard et notre version améliorée à l'aide de *CombinedMontMul*. Cependant, les termes exprimés en ℓ et 2^γ sont diminués d'environ $\cong 25\%$.

Enfin, dans le cas des méthodes dites *Regular left-to-right 2^γ-ary Exponentiation*, les termes les plus significatifs sont ceux en $\gamma\ell$ et en 2^γ. Ils sont globalement équivalents dans les deux cas, version standard et notre version améliorée à l'aide de *MultByComOp*. Cependant, les termes en ℓ sont quant à eux réduits de $\cong 50\%$.

Exemple 4.2.1. *Examinons le cas d'un module N de taille 2048 bits. Cette taille est usuelle pour les protocoles basés sur RSA. Dans la table 4.7, nous fournissons les complexités pour les trois cas examinés ci-dessus. Les améliorations sont de 13% pour l'échelle binaire de Montgomery, 4% avec l'algorithme* Regular right-to-left 2^γ-ary Exponentiation *et 8% avec l'algorithme* Regular left-to-right 2^γ-ary Exponentiation. *Pour l'échelle binaire de Montgomery avec notre approche, on note aussi la complexité inférieure en nombre d'opérations ADD de l'ordre de 1 % et supérieure de 1 % en nombre d'opérations MUL, par rapport à l'algorithme* Square-always *de Clavier* et al. *dans [15].*

TABLE 4.6 – Comparaison des complexités des algorithmes réguliers d'exponentiation modulaire

	ADD	MUL
Square-always, Clavier *et al.* [15]	$t(6n^2+\frac{55}{4}n-2)+8n^2+4n-2$	$t(3n^2+3n)+4n^2+2n$
Mont-ladder	$t(7n^2+7n-2)+8n^2+4n-2$	$t(\frac{7n^2}{2}+\frac{7n}{2}-1)+4n^2+2n$
Mont-ladder avec CMM	$t(6n^2+9n+1)+(8n^2+8n)$	$t(3n^2+4n+3)+(4n^2+4n+2)$
Right-to-left	$\gamma\ell(3n^2+5n-1)+\ell(4n^2+2n-1)$ $+2^\gamma(8n^2+4n-2)$	$\gamma\ell(\frac{3n^2}{2}+\frac{5n}{2}-1)+\ell(2n^2+n)$ $+2^\gamma(4n^2+2n)$
Right-to-left avec CMM	$\gamma\ell(3n^2+7n)+\ell(3n^2+2n+1)$ $+2^\gamma(6n^2+9n+1)+4n^2-2n-2$	$\gamma\ell(\frac{3n^2}{2}+\frac{7n}{2})+\ell(\frac{3n^2}{2}+\frac{n}{2}+3)$ $+2^\gamma(3n^2+4n+3)+2n^2-2$
Left-to-right	$\gamma\ell(3n^2+5n-1)+\ell(4n^2+2n-1)$ $+2^\gamma(4n^2+2n-1)+4n^2+2n-1$	$\gamma\ell(\frac{3n^2}{2}+\frac{5n}{2}-1)+\ell(2n^2+n)$ $+2^\gamma(2n^2+n)+2n^2+n$
Left-to-right avec MBCO	$\gamma\ell(3n^2+7n)+\ell(2n^2+5n+1)$ $+2^\gamma(4n^2+4n)+6n^2+3n-1$	$\gamma\ell(\frac{3n^2}{2}+\frac{7n}{2})+\ell(n^2+2n+2)$ $+2^\gamma(2n^2+2n+1)+3n^2+2n$

CMM=*CombinedMontMul*, MBCO=*MulByComOp*

TABLE 4.7 – Comparaison des complexités de l'exponentiation modulaire pour 2048 bits

	SqA	ML	ML CMM	R-to-L	R-to-L CMM	L-ro-R	L-to-R MBCO
#ADD/10^3	13479	15143	13183	8595	8253	8466	7804
Amélioration		12.9%		4%		7.9%	
#MUL/10^3	6488	7506	6564	4296	4120	4232	3896
Amélioration		12.6%		4.1%		7.9%	

SqA = *Square-always*, ML=Montgomery-ladder, R-to-L= Right-to-left,
L-to-R=Left-to-right, CMM=*CombinedMontMul*, MBCO=*MulByComOp*

4.2.5 Implantations logicielles, résultats et conclusion

Nous avons réalisé des implantations logicielles pour les trois algorithmes présentés dans cette section. Les principales caractéristiques de ces implantations sont les suivantes :

— écriture en langage C, compilateur `gcc 4.8.2` ;
— plate-forme Intel Core i7-4770 CPU @ 3.4GHz ;
— opérations élémentaires multiprécision de la bibliothèque `GMP 6.0.0` [2], en particulier la multiplication $1 \times n$ mots et l'addition multiprécision ;
— options *Turbo-Boost* et *Hyperthreading* désactivées pour des mesures de performances aussi précises que possible.

Les résultats des expérimentations sont reportés dans la table 4.8. Ils résultent de moyennes sur plusieurs centaines de jeux de données différents.

En synthèse de cette table, nous notons :

— Les performances constatées montrent une amélioration significative dans le cas de l'échelle binaire de Montgomery, pour toutes les tailles de modules que nous avons testées, de 9,0 % à près de 15 %.
— Toujours pour l'échelle binaire de Montgomery, notre approche se montre compétitive en comparaison de celle de Clavier *et al.* dans [15] pour les tailles de 1024 bits (gain de 10,5 %) et 2048 bits (gain de 1,9 %). Cependant, pour la taille de module de 4096 bits,

TABLE 4.8 – Performances des implantations (10^3 cycles)

Algorithme	1024 bits		2048 bits		4096 bits	
	#CC/10^3	Am. ratio	#CC/10^3	Am. ratio	#CC/10^3	Am. ratio
Square-always, Clavier *et al.* [15]	3123		19133		127293	
Éch. Montgomery	3068	9,0%	20643	9,1%	153443	14,6%
Éch. Montgomery avec CMM	2793		18773		131011	
Right-to-left	1857	0,1%	11796	1,7%	87081	4,0%
Right-to-left avec CMM	1855		11596		83599	
Left-to-right	1858	-1%	11734	3,1 %	83354	8,3 %
Left-to-right avec MBCO	1877		11368		77745	

notre approche est moins performante d'environ 2,9 %.

— Pour ce qui est de l'algorithme *Regular right-to-left* 2^γ*-ary Exponentiation*, nous observons un taux d'amélioration moins important, comme prévu par l'étude de la complexité. Néanmoins, l'amélioration constatée pour la plus grande taille testée ($4,0\%$ pour 4096 bits) est proche de l'évaluation théorique.

— Enfin, l'algorithme *Regular left-to-right* 2^γ*-ary Exponentiation* est amélioré également dans les cas de tailles de modules de 2048 et 4096 bits. On remarque également dans ce cas que l'influence du stockage en mémoire (256kO pour 2048 bits et 1MO pour 4096 bits) ne rend pas l'approche ineffective. Ce stockage est inférieur à la taille de la mémoire cache L3 dans tous les cas (8 MO pour notre plate-forme).

Chapitre 5

Combiner les opérations dans la multiplication scalaire de point de courbe elliptique sur $\mathbb{F}_{2^m}$

Le chapitre précédent a présenté des opérations combinées appliquées à l'exponentiation modulaire, dans le but d'accroître les performances. Dans le cas de la multiplication scalaire de point de courbe elliptique, les opérations de doublements et d'additions de points peuvent prendre avantage d'opérations combinées. Pour ne montrer qu'un exemple, rappelons d'abord les formules de doublement de points sur $\mathbb{F}_{2^m}$ en coordonnées mixées *PL* (voir section 2.3.3.1.2, page 63) :

$$2 \cdot (X : Y : Z : T) = (X_1 : Y_1 : Z_1 : T_1), \text{ avec :}$$

$$\begin{cases} X_1 = [A \times A + \underbrace{b \cdot T^2}_{AB,AC}], \\ Y_1 = \overbrace{B \times (B + X_1 + Z_1) + \underbrace{b \times T_1}_{AB,AC}}^{AB+CD} + T_1, \quad \text{où} \quad \begin{cases} A = X^2, \\ B = [Y]^2. \end{cases} \\ Z_1 = T \cdot A, \\ T_1 = Z_1^2, \end{cases}$$

On repère une opération combinée AB, AC du fait de l'opérande commun b dans les deux multiplications $b \times T^2$ et $b \times T_1$. Ceci est similaire dans sa forme avec ce que nous avions optimisé dans le chapitre précédent. On repère ce type d'opération ainsi que d'autres du type $AB + CD$ dans les différentes opérations sur points dans les différents systèmes de coordonnées. Par conséquent, nous modifions les algorithmes pour ces opérations sur points pour diminuer leur complexité en utilisant ces opérations de type AB, AC et $AB + CD$.

Nous proposons d'optimiser ces opérations combinées. La première optimisation de deux multiplications AB, AC avec un opérande commun A est proposée par Avanzi dans [6] pour l'algorithme *CombMul* (l'algorithme 1.12 page 40) permettant d'éviter un pré-calcul de table. Nous étendons cette idée à l'algorithme *KaratRec*. Nous proposons aussi une nouvelle optimisation basée sur les opérations de type $AB + CD$. Pour ces opérations, nous montrerons

que :

— dans le cas de la multiplication de *CombMul*, avec des opérandes de tailles n mots machine, nous pouvons économiser $60n$ WShifts and $30n$ WXORs.

— dans le cas de la multiplication basée sur l'approche de Karatsuba, la complexité peut également être réduite et nous chiffrerons l'amélioration.

Nous nous penchons sur l'implantation logicielle de la multiplication scalaire de points de courbe elliptique définie sur corps binaires $\mathbb{F}_{2^m}$ munie des opérations sur points bénéficiant des optimisations des opérations combinées étudiées.

Ce travail a été présenté au Caire le 22 juin 2013, lors de la conférence AfricaCrypt, et a été publié dans les actes de cette conférence [51].

Les éléments du corps $\mathbb{F}_{2^m}$ sont des polynômes binaires de degré au plus $m-1$, et $m \in [160, 600]$ est un nombre premier. Dans la table 5.1, nous donnons les valeurs de n et μ paquets de quatre bits pour les corps de tailles $m = 233$ et $m = 409$ que nous avons considérés dans notre travail.

TABLE 5.1 – Polynômes irréductibles et tailles des éléments de corps en nombre de mots machine et paquets de 4 bits (représentant des polynômes de degré au plus 3)

m le degré du corps	Polynôme irréductible	n (mots de 64-bit)	Nombre de paquets μ (découpage/paquets de 4 bits)
233	$x^{233} + x^{74} + 1$	4	59
409	$x^{409} + x^{87} + 1$	7	103

Dans la suite de cette section, nous présentons en premier lieu les optimisations AB, AC et $AB + CD$ dans le contexte de la multiplication de Lopez-Dahab (*CombMul*, l'algorithme 1.12 page 40) ; puis nous présentons ces optimisations dans le contexte de la multiplication utilisant l'approche de Karatsuba et le multiplieur *carry-less* (*KaratRec*, l'algorithme 1.13 page 41) ; nous appliquerons ces optimisations à différents algorithmes de multiplication scalaire de points de courbe elliptique et nous exposons pour finir les caractéristiques et les résultats des implantations logicielles.

5.1 Optimisations AB, AC et $AB + CD$ dans le cas de la multiplication *CombMul*

Nous avons présenté la multiplication *CombMul* dans l'algorithme 1.12 page 40. Nous nous appuyons sur cet algorithme pour présenter les optimisations.

5.1.1 Optimisation AB, AC

Cette approche a été proposée par Avanzi dans [6]. Dans l'algorithme 1.12 page 40, nous calculons étapes 1 à 6 les éléments de la table T, soient les seize produits $T[u] = u \cdot A$, où u sont des polynômes de degré au plus 3. Comme nous calculons deux multiplications avec le même opérande A, nous ne calculons cette table qu'une seule fois. Cela permet d'économiser $14n - 7$ WXORS et $14n - 7$ WShifts dans le calcul de AC. Ceci est montré dans l'algorithme 5.1. Cette analyse de complexité est détaillée pas à pas dans la table 5.2.

Algorithme 5.1 *CombMul_ABAC*, Avanzi dans [6]

Require: $A(x)$, $B(x)$ et $C(x)$ polynômes binaires de degré $< 64n$, et $B(x) = \sum_{j=0}^{n-1}\sum_{k=0}^{15} B_{16j+k}x^{4k+64j}$ et $C(x) = \sum_{j=0}^{n-1}\sum_{k=0}^{15} C_{16j+k}x^{4k+64j}$ décomposés en mots de 64 bits et paquets de 4 bits.

Ensure: $R(x) = A(x) \cdot B(x)$ et $R'(x) = A(x) \cdot C(x)$

// Calcul préalable de la table T telle que $T[u] = u(x) \cdot A(x)$ pour tout u tel que $\deg u(x) < 4$

1: $T[0] \leftarrow 0$;
2: $T[1] \leftarrow A$;
3: **for** k **from** 1 **to** 7 **do**
4: $\quad T[2k] \leftarrow T[k] << 1$;
5: $\quad T[2k+1] \leftarrow T[2k] \oplus A$;
6: **end for**

// décalages à gauche et accumulations

7: $R \leftarrow 0, R' \leftarrow 0$
8: **for** k **from** 15 **downto** 0 **do**
9: $\quad R \leftarrow R << 4, R' \leftarrow R' << 4$
10: $\quad$ **for** j **from** $n-1$ **downto** 0 **do**
11: $\quad\quad R \leftarrow R \oplus (T[B_{16j+k}] << 64j)$
12: $\quad\quad R' \leftarrow R' \oplus (T[C_{16j+k}] << 64j)$
13: $\quad$ **end for**
14: **end for**
15: **return** (R et R')

5.1.2 Optimisation $AB + CD$

Cette optimisation est réalisée en effectuant l'addition finale $(AB) + (CD)$ lors de l'accumulation étape 11 de l'algorithme *CombMul*. Ainsi, le calcul préalable de la table $T[u] = u \cdot A$ et $S[u] = u \cdot C$ pour les polynômes u de degré au plus 3 est inchangé. En revanche, nous accumulons $T[B_{16j+k}]$ et $S[D_{16j+k}]$ dans la même variable $R \leftarrow R \oplus (T[B_{16j+k}] \oplus S[B_{16j+k}])x^{64j}$. Le décalage à gauche de 4 bits n'est effectué qu'une seule fois sur R. Cette démarche est présentée dans l'algorithme 5.2.

La complexité de l'algorithme 5.2 peut se déduire de la complexité de l'algorithme 1.12 *CombMul* (voir la table 1.8 page 40) :

— dans l'algorithme *CombMul_ABplusCD*, nous avons le calcul préalable de deux tables (T et S), soit deux fois la complexité d'une seule table ;

TABLE 5.2 – Complexité de l'algorithme 5.1 *CombMul_ABAC*

	Opérations	#WXOR	#WShift	#WAND
7 étapes 4	$T[k] << 1$	$7 \times (n-1)$	$7 \times (2n-1)$	-
7 étapes 5	$T[2k] \oplus A$	$7 \times n$	-	-
15 étapes 9	$R << 4, R' << 4$	$60 \times n$	$120 \times n$	-
$16 \times n$ étapes 11	$T[B_{16j+k}] << 64j$ $R \oplus (T[B_{16j+k}] << 64j)$	- μn	$\mu - n$ -	μ -
$16 \times n$ étapes 12	$T[C_{16j+k}] << 64j$ $R' \oplus (T[C_{16j+k}] << 64j)$	- μn	$\mu - n$ -	μ -
Total		$2\mu n + 74n - 7$	$2\mu + 132n - 7$	2μ

Algorithme 5.2 *CombMul_ABplusCD*(A,B,C,D)

Require: Quatre polynômes binaires A, B, C et D de degré $< 64n$, et $B(x) = \sum_{j=0}^{n-1} \sum_{k=0}^{15} B_{16j+k} x^{4k+64j}$ avec $\deg B_{16j+k} < 4$ et $D(x) = \sum_{j=0}^{n-1} \sum_{k=0}^{15} D_{16j+k} x^{4k+64j}$ avec $\deg D_{16j+k} < 4$.

Ensure: $R(x) = A(x) \cdot B(x) + C(x) \cdot D(x)$ // Calcul préalable des tables T et S telles que $T[u] = u(x) \cdot A(x)$ et $S[u] = u(x) \cdot C(x)$ pour tout $\deg u(x) < 4$

1: $T[0] \leftarrow 0$, $S[0] \leftarrow 0$
2: $T[1] \leftarrow A$, $S[1] \leftarrow C$
3: **for** k **from** 1 **to** 7 **do**
4: $\quad T[2k] \leftarrow T[k] << 1$, $S[2k] \leftarrow S[k] << 1$
5: $\quad T[2k+1] \leftarrow T[2k] \oplus A$, $S[2k+1] \leftarrow S[2k] \oplus C$
6: **end for**// décalages à gauche et accumulations
7: $R \leftarrow 0$
8: **for** k **from** 15 **downto** 0 **do**
9: $\quad R \leftarrow R << 4$
10: $\quad$ **for** j **from** $n-1$ **downto** 0 **do**
11: $\quad\quad R \leftarrow R \oplus (T[B_{16j+k}] \oplus S[D_{16j+k}]) << 64j$
12: $\quad$ **end for**
13: $\quad$ **return** (R)
14: **end for**

— Les accumulations $R \leftarrow R \oplus (T[B_{16j+k}] \oplus S[D_{16j+k}])x^{64j}$ représentent le double de celle montrée pour l'accumulation dans l'algorithme *CombMul* (étape 11).
— La quantité de décalages $R \leftarrow R << 4$ est inchangée par rapport à l'algorithme *CombMul*.

La complexité est analysée pas à pas dans la table 5.3.

TABLE 5.3 – Complexité de l'algorithme 5.2 (*CombMul_ABplusCD*)

	Opérations	#WXOR	#WShift	#WAND
7 étapes 4	$T[k] << 1, S[k] << 1$	$14 \times (n-1)$	$14 \times (2n-1)$	-
7 étapes 5	$T[2k] \oplus A, S[2k] \oplus C$	$14 \times n$	-	-
15 étapes 9	$R << 4$	$30 \times n$	$60 \times n$	-
$16 \times n$ étapes 11	$(T[B_{16j+k}] \oplus S[D_{16j+k}]) << 64j$	μn	$2\mu - 2n$	2μ
	$R \oplus (T[B_{16j+k}] \oplus S[D_{16j+k}]) << 64j$	μn	-	-
Total		$2\mu n + 58n - 14$	$2\mu + 86n - 14$	2μ

5.2 Optimisations AB, AC et $AB + CD$ dans le cas de l'approche *KaratRec*

L'optimisation basée sur AB, AC peut être appliquée au cas de l'algorithme *KaratRec* (voir l'algorithme 1.13 page 41). En effet, dans cet algorithme récursif, l'addition des deux moitiés $A_0 + A_1$ n'est effectuée qu'une seule fois pour l'opérande A. Ceci est montré dans l'algorithme 5.3.

L'optimisation basée sur l'opération combinée $AB + CD$ est aussi adaptée de la façon suivante : l'addition est effectuée avant la reconstruction des deux produits AB et CD, ce qui signifie que nous n'avons qu'une seule reconstruction récursive au lieu de deux. Cette approche se voit dans l'algorithme 5.4.

Algorithme 5.3 *KaratRec_ABAC*(A,B,C,n)

require : A, B et C polynômes représentés par $n = 2^s$ mots machine de 64 bits.
ensure : $R = A \cdot B$ and $S = A \cdot C$
if $n = 1$ **then**
return$(Mul64(A, B), Mul64(A, C))$
else
// Séparation en deux moitiés de $n/2$ mots machine.
$A = A_0 + x^{64n/2}A_1,\ B = B_0 + x^{64n/2}B_1,$
$C = C_0 + x^{64n/2}C_1$
// Additions des moitiés
$A_2 = A_0 + A_1,\ B_2 = B_0 + B_1,\ C_2 = C_0 + C_1$
// Multiplications récursives
$R_0, S_0 \leftarrow$ *KaratRec_ABAC*$(A_0, B_0, C_0, n/2)$
$R_1, S_1 \leftarrow$ *KaratRec_ABAC*$(A_1, B_1, C_1, n/2)$
$R_2, S_2 \leftarrow$ *KaratRec_ABAC*$(A_2, B_2, C_2, n/2)$
// Reconstruction
$R \leftarrow R_0 + (R_0 + R_1 + R_2)x^{64n/2} + R_1x^{64n}$
$S \leftarrow S_0 + (S_0 + S_1 + S_2)x^{64n/2} + S_1x^{64n}$
return(R, S)
end if

Algorithme 5.4 *KaratRec_ABpCD*(A,B,C,D,n)

require : A, B, C et D polynômes représentés par $n = 2^s$ mots machine de 64 bits.
ensure : $R = AB + CD$
if $n = 1$ **then**
return$(Mul64(A, B) + Mul64(C, D))$
else
// Séparation en deux moitiés de $N/2$ mots machine.
$A = A_0 + x^{64n/2}A_1,\ B = B_0 + x^{64n/2}B_1,$
$C = C_0 + x^{64n/2}C_1,\ D = D_0 + x^{64n/2}D_1$
// Additions des moitiés
$A_2 = A_0 + A_1,\ B_2 = B_0 + B_1$
$C_2 = C_0 + C_1,\ D_2 = D_0 + D_1$
// Multiplications/additions (récursives)
$R_0 \leftarrow$ *KaratRec_ABpCD*$(A_0, B_0, C_0, D_0, n/2)$
$R_1 \leftarrow$ *KaratRec_ABpCD*$(A_1, B_1, C_1, D_1, n/2)$
$R_2 \leftarrow$ *KaratRec_ABpCD*$(A_2, B_2, C_2, D_2, n/2)$
// Reconstruction
$R \leftarrow R_0 + (R_0 + R_1 + R_2)x^{64n/2} + R_1x^{64n}$
return(R)
end if

5.2.1 Complexité de *KaratRec_ABAC*.

Dans la première récurrence, nous avons $3n/2$ WXORs pour $A_0 + A_1,\ B_0 + B_1$ et $C_0 + C_1$ plus $6n$ WXORs pour la reconstruction de R et S. Ceci conduit à la complexité suivante :

$$\begin{cases} \#WXOR(n)=15n/2 + 3\,\#\text{WXOR}(n/2), \\ \#WXOR(1)=0. \end{cases} \implies \#WXOR(n) = 15n^{\log_2(3)} - 15n$$

$$\begin{cases} \#Mult64(n)=3\#Mult64(n/2), \\ \#Mult64(1)=2. \end{cases} \implies \#Mult64(n) = 2n^{\log_2(3)}.$$

5.2.2 Complexité de *KaratRec_ABpCD*.

Dans la première récurrence, nous avons $2n$ WXORs dans le calcul de $A_0 + A_1,\ B_0 + B_1,\ C_0 + C_1$ et $D_0 + D_1$ plus $6n/2$ WXORs pour la reconstruction de R. La complexité pour $n = 1$ est égale à $2Mult64$ plus un WXOR. Sur cette base, nous en déduisons la complexité pour l'algorithme *KaratRec_ABpCD* :

$$\#WXOR(n) = 10n^{\log_2(3)} - 10n, \qquad \#Mult64(n) = 2n^{\log_2(3)}.$$

5.3 Complexités et performances

À l'aide des complexités précédentes, nous pouvons maintenant calculer les complexités des algorithmes de multiplications optimisées AB, AC et $AB + CD$ dans le cas de polynômes représentant des éléments du corps $\mathbb{F}_{2^m}$ avec $m = 233$ et $m = 409$. Nous avons également réalisé une implantation logicielle de ces algorithmes sur plate-formes Intel Core 2 et Intel Core i5. Notre implantation utilise les registres 128 bits et les instructions vectorielles disponibles sur ces deux processeurs. Sur les Core 2, nous utilisons l'algorithme *CombMul* dans sa version proposée dans [9, 4] qui repose principalement sur l'utilisation de décalages d'un nombre de bits multiple de 8. Ceci est plus efficace que des décalages d'un nombre arbitraire de bits dans le cas de données stockées sur 128 bits. Sur Core i5, nous avons implanté l'approche récursive

de Karatsuba (multiplication *KaratRec*) et nous utilisons l'instruction de multiplication *carry-less* dénommée PCLMUL, qui multiplie deux mots de 64 bits stockés dans un registre 128 bits.

Nous comparons ces complexités et performances avec les versions classiques des mêmes algorithmes. Les résultats de ces complexités sont reportés dans la table 5.5.

TABLE 5.4 – Complexité/performance de l'approche *CombMul* optimisée sur Core 2 (2,5 GHz)

Algorithme	Complexité totale en nombre d'opérations sur mots	233		409	
		#W.Op.	#CC	#W.Op.	#CC
CombMul	$n\mu + 2\mu + 117n - 14$	808	336	1732	795
CombMul_ABAC	$2n\mu + 4\mu + 206n - 14$	1511	555	3282	1597
CombMul_ABplusCD	$2n\mu + 4\mu + 144n - 28$	1256	564	2834	1737

#W. Op. = nombre d'opérations élémentaires (sur mots, WXOR, WAND, WShift).
#CC = nombre de cycles d'horloge.

TABLE 5.5 – Complexité/performance de l'approche *KaratRec* optimisée sur Core i5 (2,5 GHz)

Algorithme	Complexité pour $n = 2^s$		233			409		
	#WXOR	#Mul64	#WXOR	#Mul64	#CC	#WXOR	#Mul64	#CC
KaratRec	$8n^{\log_2(3)} - 8n$	$n^{\log_2(3)}$	40	9	107	152	27	286
KaratRec_ABAC	$15n^{\log_2(3)} - 15n$	$2n^{\log_2(3)}$	75	18	189	285	54	566
KaratRec_ABpCD	$10n^{\log_2(3)} - 10n$	$2n^{\log_2(3)}$	50	18	182	190	54	541

Sur la base de ces résultats, nous remarquons que l'optimisation $AB + CD$ est toujours meilleure que l'optimisation AB, AC et meilleure que deux multiplications effectuées indépendamment. Sur les performances, nous notons que :

— Sur Core 2, l'optimisation $ABplusCD$ est toujours plus lente que AB, AC. De plus, les optimisations $ABplusCD$ et AB, AC seulement dans le cas $m = 233$, étant plus rapides que deux multiplications effectuées indépendamment dans ce cas. Ceci contredit les complexités correspondantes, qui présentent des différences plus importantes.

— Sur Core i5, les résultats de performances sont mieux corrélés avec les valeurs de complexités : pour les deux corps considérés, $ABplusCD$ et AB, AC sont plus rapides que deux multiplications effectuées indépendamment et $ABplusCD$ est toujours meilleur que AB, AC.

Dans la littérature, nous trouvons des valeurs de performances pour des implantations de l'algorithme *CombMul* sur Core 2 dans [4]. Les auteurs dans [4] fournissent des performances de l'ordre de $[241, 276]$ cycles d'horloge pour la multiplication de polynômes de taille $m = 233$ et de l'ordre de $[690, 751]$ pour $m = 409$, deux valeurs meilleures que ce que nous fournissons dans la table 5.4.

Les résultats de performances que nous présentons sur Core i5 sont en revanche meilleurs que ceux fournis dans Taverne *et al.* [64] : 128 cycles d'horloge pour $m = 233$ et 345 cycles d'horloge pour $m = 409$. Les mêmes auteurs ont depuis mis à jour leurs valeurs dans [65] et revendiquent maintenant 100 cycles d'horloge pour $m = 233$ et 270 cycles d'horloge pour $m = 409$ (mêmes plate-forme et compilateur).

5.4 Implantations de la multiplication scalaire basée sur les optimisations AB, AC et $AB + CD$

Dans cette section, nous présentons les résultats expérimentaux de nos implantations logicielles basées sur les optimisations AB, AC et $AB + CD$ présentées précédemment. Nous passons en revue en premier lieu les meilleures formules pour les opérations sur points de courbe elliptique et nous décrivons comment nous tirons parti des optimisations AB, AC et $AB + CD$ dans ces formules. Nous présentons ensuite les stratégies utilisées dans nos implantations : algorithmes de multiplication scalaire et implantation des opérations sur le corps (carré, inversion...) Nous présentons pour finir les résultats de performances sur Intel Core 2 et Intel Core i5.

5.4.1 Arithmétique des courbes elliptiques

Nous considérons deux courbes elliptiques préconisées dans les normes du NIST [19] : $B233$ et $B409$. Il s'agit de courbes ordinaires définies par l'équation de Weierstrass

$$y^2 + xy = x^3 + x^2 + b \text{ où } b \in \mathbb{F}_{2^m}.$$

5.4.1.1 Optimisation AB, AC et $ABplusCD$ dans les opérations de points de courbe elliptique

Nous appliquons nos optimisations aux opérations réalisées dans le systèmes de coordonnées *PL* (voir section 2.3.3.1.2, page 63). Ce système représente le point P à l'aide des coordonnées projectives $P = (X : Y : Z : T)$. Nous notons, comme en section 2.3.3.1.2, la multiplication sans réduction $A \times B$ et $[R]$ représente la réduction du polynôme R modulo le polynôme irréductible définissant le corps $\mathbb{F}_{2^m}$.

5.4.1.1.1 Doublement en coordonnées *PL*.

Nous calculons le doublement $P_1 = (X_1 : Y_1 : Z_1 : T_1) = 2 \cdot (X : Y : Z : T)$ du point $P = (X : Y : Z : T)$ en effectuant la séquence d'opérations suivante :

$$A{=}X^2, \quad B{=}[Y]^2.$$

puis :

$$Z_1{=}[T \times A], \quad T_1{=}[Z_1^2], \quad X_1{=}[A^2 + \underbrace{b \times T^2}_{AB,AC}], \quad Y_1{=}\overbrace{B \times (B + X_1 + Z_1) + \underbrace{b \times T_1}_{AB,AC}}^{ABplusCD} + T_1.$$

5.4.1.1.2 Addition de points en coordonnées *PL*.

Reprenons les formules fournies par *PL* dans [33] : $P_1 = (X_1 : Y_1 : Z_1 : T_1)$ est représenté en coordonnées *PL* et $P_2 = (X_2 : Y_2 : 1 : 1)$ en coordonnées affines, c'est-à-dire $Z_2 = T_2 = 1$. Nous obtenons $P_3 = (X_3 : Y_3 : Z_3 : T_3)$ en effectuant la séquence d'opérations suivantes :

$$A{=}X_1 + [X_2 \cdot Z_1], \quad B{=}[Y_1 + Y_2 \cdot T_1], \quad C{=}[A \cdot Z_1], \quad D{=}\underbrace{[C \cdot (B + C)]}_{AB,AC}.$$

On en déduit $Z_3 = [C^2],\ T_3 = [Z_3^2],$ et

$$X_3{=}[B^2 + \underbrace{C \cdot [A^2]}_{AB,AC}] + D, \quad Y_3{=}\overbrace{[(X_3 + [X_2 \cdot Z_3]) \cdot D + (X_2 + Y_2) \cdot T_3]}^{ABplusCD}.$$

Dans les formules ci-dessus, nous indiquons les opérations qui peuvent être effectuées à l'aide de l'optimisation $AB + CD$ et celles qui tirent parti de l'optimisation AB, AC.

5.4.1.1.3 Optimisation AB, AC et $ABplusCD$ dans le système de coordonnées projectives Lopez-Dahab $\mathcal{LD}$.

Doublement

Les formules de doublement en coordonnées $\mathcal{LD}$ (voir la table 2.8 page 63) permettent une application de l'optimisation $AB + CD$ dans le calcul de Y_1 :

$$2 \cdot (X : Y : Z) = (X_1 : Y_1 : Z_1), \text{ avec } \begin{cases} X_1 = X^4 + b \cdot Z^4, \\ Y_1 = \overbrace{bZ^4 \cdot Z_1 + X_1 \cdot (aZ_1 + Y^2 + bZ^4)}^{ABplusCD}, \\ Z_1 = X^2 \cdot Z^2. \end{cases}$$

Addition en coordonnées mixées

De même, dans les formules d'addition mixée (voir la table 2.9 page 63), le calcul des paramètres F et G est modifié pour permettre deux applications de l'optimisation $AB + CD$, ainsi qu'une application de l'optimisation AB, AC :

$$(X_1 : Y_1 : Z_1) + (X_2 : Y_2 : 1) = (X_3 : Y_3 : Z_3),$$

$$\text{avec } \begin{cases} X_3 = A^2 + D + E, \\ Y_3 = \underbrace{X_3 \cdot F + \overbrace{Z_3 \cdot G}^{AB,AC}}_{ABplusCD}, \\ Z_3 = C^2, \end{cases} \quad \text{où } \begin{cases} A = Y_2 \cdot Z_1^2 + Y_1, \\ B = X_2 \cdot Z_1 + X_1, \\ C = Z_1 \cdot B, \\ D = B^2 \cdot (C + aZ_1^2), \\ E = A \cdot C, \\ F = E + Z_3, \\ G = \underbrace{E \cdot X_2 + \overbrace{Y_2 \cdot Z_3}^{AB,AC}}_{ABplusCD}. \end{cases}$$

5.4.1.1.4 Optimisation AB, AC et $ABplusCD$ dans le système de coordonnées projectives XZ pour l'échelle binaire de Montgomery.

Dans le corps de l'algorithme 3.6, l'échelle binaire de Montgomery avec pseudo-addition, étapes 4 et 7, nous appliquons l'optimisation $ABplusCD$ dans le calcul de X_1 et X_2 :

1: $X_1 \leftarrow x, Z_1 \leftarrow 1, X_2 \leftarrow x^4 + b, Z_2 \leftarrow x^2$
2: **for** $i = t - 2$ downto 0 **do**
3: **if** $(k_i = 0)$ **then**
4: $T \leftarrow Z_1, Z_1 \leftarrow (X_1 Z_2 + X_2 Z_1)^2, X_1 \leftarrow \overbrace{xZ_1 + X_1 X_2 T Z_2}^{ABplusCD}$
5: $T \leftarrow X_2, X_2 \leftarrow X_2^4 + bZ_2^4, Z_2 \leftarrow T^2 + Z_2^2$
6: **else**
7: $T \leftarrow Z_2, Z_2 \leftarrow (X_1 Z_2 + X_2 Z_1)^2, X_2 \leftarrow \overbrace{xZ_2 + X_1 X_2 T Z_1}^{ABplusCD}$
8: $T \leftarrow X_1, X_1 \leftarrow X_1^4 + bZ_1^4, Z_1 \leftarrow T^2 + Z_1^2$
9: **end if**
10: **end for**

Nous n'appliquons pas l'optimisation $ABplusCD$ au calcul de Z_1 ou Z_2, car le résultat de la multiplication $X_1 Z_2$ est récupéré pour le calcul de X_1 et X_2.

5.4.1.1.5 Cas de l'opération de *halving*

Enfin, concernant l'opération de *halving* (voir section 2.3.3.2 page 64), l'application des optimisations AB, AC ou $ABplusCD$ n'est pas possible. En effet, cette opération consiste en un calcul de *Half-trace*, suivie par une seule multiplication, un calcul de trace et une racine carrée.

5.4.1.2 Algorithmes de multiplication scalaire

Soit $P \in E(\mathbb{F}_{2^m})$ et k un entier de taille t-bits. La multiplication scalaire $k \cdot P$ sur la courbe $E(\mathbb{F}_{2^m})$ a été implantée de la façon suivante :

— *Double-and-add*
Nous utilisons l'algorithme 2.8 présenté page 70, avec une taille de fenêtre $w = 4$. Nous calculons donc les $T[i] = i \cdot P$ pour les entiers impairs $0 < i < 2^{w-1}$ au début de la multiplication scalaire. Nous utilisons les formules de doublement et d'addition en coordonnées *PL* et $\mathcal{LD}$.

— *Halve-and-add, Parallel (Double-and-add, Halve-and-add)*
Cette approche est présentée dans l'algorithme 2.10 page 72. L'opération de *Halving* est décrite section 2.3.3.2 page 64.

— *Échelle binaire de Montgomery*
Nous implantons ici l'échelle binaire de Montgomery, soit l'algorithme 3.6 page 85.

5.4.2 Stratégies d'implantation

Nous traitons ici du détail des stratégies adoptées pour l'implantation des opérations sur le corps $\mathbb{F}_{2^m}$. Nous passons en revue chacune d'elles.

— *Multiplication :* nous nous basons sur les algorithmes de multiplications décrits section 5.3. Quelques spécificités doivent cependant être mentionnées. Sur plate-forme Intel Core 2, nous utilisons la version *CombMul* pour la multiplication telle que décrite dans [9, 4], utilisant les registres et jeu d'instruction 128 bits (`MMX`, `AES`). Sur plate-forme Intel Core i5, les mêmes jeux d'instructions vectoriels 128 bits dans l'implantation de l'approche de Karatsuba sont utilisés, avec l'instruction de la multiplication *carry-less* qui effectue la multiplication de polynômes de taille 64 bits.

— *Élévation au carré :* pour l'élévation au carré, nous utilisons la stratégie décrite dans [4]. Nous stockons dans un mot de 128 bit Sq chacun des carrés de polynômes de taille de 4 bits. Ainsi, dans chaque mot de 128 bits $A[i]$ de A, nous séparons les morceaux de 4

bits pairs et impairs à l'aide d'un masquage et d'un décalage et utilisons l'instruction `_mm_shuffle_epi8` (intrinsic disponible en langage C avec l'option de compilation `-mavx`). La première entrée de cette instruction est Sq et la deuxième entrée est le mot contenant les paquets de 4 bits pairs et impairs de $A[i]$. Le résultat est un mot de 128 bits contenant le carré de chacun de ces morceaux. Les octets sont alors réordonnés dans deux mots de 128 bits. Le lecteur peut se reporter *Algorithm 1* dans [4] pour plus de détails.

— *Racine carrée :* le calcul de la racine carrée est implanté selon la stratégie décrite section 1.2.3.2 (l'algorithme 1.15 page 43) et s'effectue selon l'expression

$$\sqrt{A} = (\sum_{i=0}^{\lceil m/2 \rceil} a_{2i} X^i) + \sqrt{x}(\sum_{i=0}^{\lceil m/2 \rceil} a_{2i+1} X^i).$$

Nous séparons les coefficients d'indice pair et impair de A à l'aide de l'instruction `_mm_shuffle_epi8` et en réordonnant les octets obtenus. Ensuite, la multiplication par $\sqrt{x}$ est effectuée à l'aide de décalages et d'additions, du fait de l'expression creuse de $\sqrt{x}$ pour les corps considérés avec $m = 233$ et $m = 409$.

— *Réduction :* nous utilisons la stratégie que nous avons décrite section 1.2.3.3 page 44. Les polynômes irréductibles sont creux pour les corps considérés (cf. la table 5.1). La réduction se limite donc à une séquence de décalages et de WXORs.

— *Inversion :* l'inversion utilise l'algorithme 1.17 présenté page 45. L'implantation des *multisquarings* fait appel à des tables calculées au préalable, comme décrit dans [64].

— *Half-trace :* dans l'opération de *halving* que nous avons présentée section 2.3.3.2 page 64, le calcul principal concerne l'évaluation de la *half-trace* (H) d'un élément du corps : $H(A) = \sum_{i=0}^{(m-1)/2} A^{2^{2i}}$. Notre implantation s'inspire encore une fois de [64] et [18] et fait usage de l'instruction intrinsic `_mm_shuffle_epi8` pour le calcul de la *half-trace* des bits pairs de A, et une table pour les bits impairs. On trouvera plus de détails dans [64, 65].

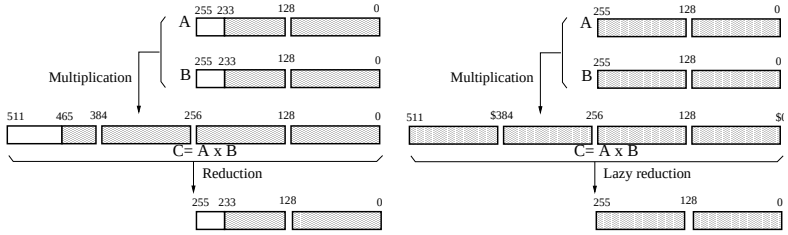
Lazy reductions. Une optimisation appelée *lazy-reduction* peut être utilisée pour accélérer les opérations de doublement et d'addition, comme décrit dans [6, 5]. Cette optimisation consiste à éliminer les réductions qui ne sont pas indispensables durant une suite de multiplications et de carrés dans les formules de doublement ou d'addition de points. Nous considérons ici les deux possibilités suivantes tirées de [6, 5] :

— *Lazy-reduction 1 (LR1) :* cette optimisation regroupe les opérations de réduction correspondant à des carrés ou des multiplications distinctes. Par exemple, dans une suite d'opérations du type $A^2 + C \cdot D$, on peut effectuer le carré A^2 et la multiplication $C \cdot D$ sans effectuer la réduction, puis réaliser l'addition des résultats de degré $2m - 2$ pour terminer par une unique réduction du résultat de cette addition. Ceci permet d'économiser une réduction au prix d'une addition supplémentaire de degré $m - 1$. Le nombre total de WXORs et WShifts s'en trouve réduit. Dans les formules de doublement et d'addition en coordonnées *PL* et $\mathcal{LD}$, mais aussi dans la pseudo-addition de l'échelle binaire de Montgomery, les crochets $[\cdot]$ dénotent un calcul sans réduction (cf. section 5.4.1). Cette optimisation est baptisée LR1.

— *Lazy-reduction 2 (LR2) :* cette optimisation consiste à ne pas effectuer une réduction totale pour obtenir un résultat de degré $m - 1$, mais d'un degré supérieur correspondant à la taille de stockage des données. Nous avons appliqué cette technique uniquement au cas $m = 233$: la réduction polynomiale est effectuée jusqu'au degré 255 au lieu de 232. Dans la multiplication utilisant l'algorithme *KaratRec* (sur plate-forme Intel Core i5), nous effectuons des multiplications de polynômes de taille 256, soit justement un degré

de 255. Ne pas effectuer la réduction des coefficients correspondant aux monômes de degré $[233, 256]$ permet d'économiser un nombre significatif de WXORs et WShifts. La figure 5.1 illustre graphiquement cette stratégie et montre comment la réduction des coefficients $c_{255}, \ldots, c_{233}$ est évitée. Cette technique est baptisée LR2.

FIGURE 5.1 – Réduction ordinaire vs *Lazy-reduction 2*



(a) Réduction ordinaire (b) *Lazy-reduction 2*

Cette stratégie ne s'applique pas au cas de l'implantation sur plate-forme Intel Core 2 utilisant l'algorithme *CombMul*, qui multiplie des polynômes de degré 232 et non 255. Dans le cas du corps binaire avec $m = 409$, l'optimisation LR2 n'apporte aucun gain dans le nombre de mots devant être réduit et nous ne l'avons donc pas appliquée.

5.4.3 Performances des implantations sur plate-forme Intel Core 2

Nos implantations ont été réalisées sur deux courbes elliptiques sur corps binaires préconisées par le NIST [19], les courbes B233 et B409. Nous donnons les paramètres de ces deux courbes en annexe 8.1.1.

Les performances de nos implantations sont transcrites dans la table 5.6. Le système d'exploitation est la distribution Linux Ubuntu 11.10, le compilateur est gcc 4.6.1. Les performances sont mesurées en nombre de cycles d'horloge, obtenu en utilisant le compteur de cycle `rdtsc` (un compteur par cœur). Les valeurs de la table 5.6 sont des moyennes sur des jeux de données générées aléatoirement.

Les résultats expérimentaux pour l'optimisation *Lazy-reduction* LR1 ne se montrent pas aussi intéressants que ce à quoi l'on pouvait s'attendre : tous les codes mettant en œuvre cette optimisation s'avèrent en effet plus lents que leurs contreparties sans cette optimisation. En conséquence, nous ne l'avons pas combinée avec les optimisations AB, AC et $ABplusCD$.

Sur la base des résultats transcrits dans la table 5.6, on peut remarquer que la technique d'optimisation $AB + CD$ apporte une amélioration significative sur la multiplication scalaire de point de courbe elliptique uniquement dans le cas du corps de taille $m = 233$. La technique AB, AC apporte une accélération sensible en comparaison des codes sans cette optimisation aussi dans le cas $m = 233$, mais pour certains algorithmes de multiplications comme le *Halve-and-add* ou *Double-and-add* avec opérations en coordonnées $\mathcal{LD}$, les performances chutent quelque peu. Dans le cas du corps $m = 409$, aucune des optimisations n'apporte d'amélioration de performances, ce qui confirme les résultats obtenus dans la table 5.4.

Nous n'avons pu trouver dans la littérature aucun résultat de performance d'implantation sur Intel Core 2 pour les mêmes courbes et corps binaires. Nous mentionnons tout de même Aranha *et al.* dans [4], qui rapportent des temps d'exécution de [785000,858000] cycles d'horloge pour la courbe du NIST B283 et [4310000,4754000] cycles d'horloge pour la courbe du NIST B571, pour une multiplication scalaire utilisant l'algorithme *Double-and-add*, sur Intel Core 2. Ces ordres de grandeurs sont une confirmation de la pertinence de nos propres résultats.

TABLE 5.6 – Performances en 10^3 cycles d'horloge $(\#CC)$ pour une multiplication scalaire de point de courbe elliptique sur Intel Core 2 (2,5 GHz)

	Optimisation	Formules	$m = 233$	$m = 409$
			$(\#CC)/10^3$	$(\#CC)/10^3$
Double-and-add	sans	$\mathcal{PL}$	592	2125
		$\mathcal{LD}$	613	2192
	LR1	$\mathcal{PL}$	1249	2207
		$\mathcal{LD}$	1179	2832
	AB, AC	$\mathcal{PL}$	558	6217
		$\mathcal{LD}$	928	2917
	$ABplusCD$	$\mathcal{PL}$	542	2187
		$\mathcal{LD}$	553	2296
Halve-and-add	sans	$\mathcal{PL}$	387	1504
		$\mathcal{LD}$	403	1575
	LR1	$\mathcal{PL}$	651	1706
		$\mathcal{LD}$	855	1837
	AB, AC	$\mathcal{PL}$	858	2277
		$\mathcal{LD}$	887	2359
	$ABplusCD$	$\mathcal{PL}$	375	1504
		$\mathcal{LD}$	386	1640
Parallèle$^{(*)}$ (*Double-and-add* + *Halve-and-add*)	sans	$\mathcal{PL}$	280	965
		$\mathcal{LD}$	295	999
	LR1	$\mathcal{PL}$	335	1042
		$\mathcal{LD}$	315	1104
	AB, AC	$\mathcal{PL}$	270	2311
		$\mathcal{LD}$	289	1362
	$ABplusCD$	$\mathcal{PL}$	273	977
		$\mathcal{LD}$	277	1014
Montgomery	sans	-	593	2190
	LR1	-	637	2482
	$ABplusCD$	-	549	2289

$(*)$ Les optimisations AB, AC et $ABplusCD$ sont apliquées sur la partie *double-and-add* uniquement.

5.4.4 Performances des implantations sur Intel Core i5

Nos implantations ont été réalisées sur les deux mêmes courbes elliptiques sur corps binaires préconisées par le NIST [19] que précédemment, les courbes B233 et B409 dont les paramètres figurent en annexe 8.1.1.

Dans la table 5.7, nous transcrivons les résultats obtenus sur Intel Core i5 utilisant les stratégies décrites plus haut dans les sections 5.4.1 et 5.4.2. Le système d'exploitation est la distribution Linux Ubuntu 12.10 et le compilateur est gcc 4.7.2. Nous avons aussi désactivé les modes *Turbo-Boost* et *Hyper Threading* du processeur Core i5 pour éviter des mesures erronées des compteurs de cycles `rdtsc`.

Nous remarquons que l'optimisation *Lazy-reduction* apporte une accélération significative, en particulier en comparaison de l'implantation conventionnelle démunie de cette optimisation. De plus, sauf rares exceptions, les optimisations $AB + CD$ et AB, AC améliorent les performances en comparaison des versions avec ou sans *Lazy-reduction* ou non. Dans le cas de la multiplication scalaire avec l'algorithme *Halve-and-add*, l'amélioration est plus modeste que dans le cas *Double-and-add*, et ceci s'explique par le fait que dans l'approche *Halve-and-add*, les optimisations n'apparaissent que dans les additions de points de courbe, qui sont beaucoup moins fréquentes que les *halvings* de points.

D'une façon générale, l'optimisation $AB+CD$ apporte une amélioration supérieure à l'optimisation AB, AC. Enfin, nous n'avons rencontré que deux cas pour lesquels aucune des deux optimisations $AB + CD$ et AB, AC n'apporte d'amélioration, c'est l'algorithme parallèle pour $m = 233$ et *Halve-and-add* pour $m = 409$.

TABLE 5.7 – Performances en 10^3 cycles d'horloge ($\#CC$) pour une multiplication scalaire de point de courbe elliptique sur Intel Core i5 (2,5 GHz)

	Optimisations	Courbe Formules	$m = 233$	$m = 409$
			$\#CC/10^3$	$\#CC/10^3$
Double-and-add	sans	*PL*KK	246	917
		$\mathcal{LD}$	252	940
	LR1 et LR2$^{(**)}$	*PL*	220	906
		$\mathcal{LD}$	228	959
	AB, AC et LR1 et LR2$^{(**)}$	*PL*	219	903
		$\mathcal{LD}$	226	961
	$ABplusCD$ et LR1 et LR2$^{(**)}$	*PL*	214	877
		$\mathcal{LD}$	222	903
Halve-and-add	sans	*PL*	165	667
		$\mathcal{LD}$	169	719
	LR1 et LR2$^{(**)}$	*PL*	150	723
		$\mathcal{LD}$	155	708
	AB, AC et LR1 et LR2$^{(**)}$	*PL*	149	733
		$\mathcal{LD}$	155	720
	$ABplusCD$ et LR1 et LR2$^{(**)}$	*PL*	150	696
		$\mathcal{LD}$	154	689
Parallèle$^{(*)}$	sans	*PL*	131	466
		$\mathcal{LD}$	133	478
	LR1 and LR2$^{(**)}$	*PL*	116	458
		$\mathcal{LD}$	122	474
	AB, AC and LR1 and LR2$^{(**)}$	*PL*	117	457
		$\mathcal{LD}$	123	476
	$ABplusCD$ and LR1 and LR2$^{(**)}$	*PL*	117	452
		$\mathcal{LD}$	118	467
Montgomery	sans	-	244	924
	LR1 and LR2$^{(**)}$	-	229	886
	$ABplusCD$ and LR1 and LR2$^{(**)}$	- -	220	883

$(*)$ Les optimisations AB, AC et $ABplusCD$ sont appliquées uniquement sur la partie *Double-and-add*.
$(**)$ L'optimisation LR2 est appliquée uniquement au cas $m = 233$

Nous pouvons comparer nos résultats brièvement avec ceux publiés par Aranha *et al.* sur Intel Core i5 avec compilateur gcc dans [65]. On remarque qu'excepté pour l'implantation parallèle avec $m = 409$, nos résultats sont compétitifs avec les performances qu'ils publient dans cet article. Cela signifie que notre implantation atteint le niveau de performances de [65] et que les optimisations que nous proposons sont efficaces lorsque elles sont implantées à côté des meilleures stratégies connues sur Intel Core i5.

5.4.5 Conclusion sur la multiplication scalaire de point de courbe elliptique

Nous avons présenté les briques de base des optimisations pour des implantations logicielles de multiplication scalaire de points de courbe elliptique. L'idée principale est de profiter des opérations combinées de type $AB + CD$ ou AB, AC lorsqu'elles apparaissent dans l'addition ou le doublement de points utilisés dans les algorithmes de multiplication scalaire de point. Dans de telles opérations combinées $AB + CD$ ou AB, AC, des calculs peuvent être mutualisés et les économies d'opérations qui en résultent conduisent à une amélioration des performances des implantations logicielles. Cette idée avait été mentionnée précédemment par Avanzi dans [6] pour l'optimisation AB, AC dans le cas de la multiplication de polynôme basée sur l'algorithme *CombMul*. Nous avons étendu cette idée aux variantes basées sur la multiplication de Karatsuba. Nous avons aussi étudié l'optimisation $AB + CD$ dans les deux cas des multiplications utilisant l'algorithme *CombMul* et l'approche récursive de Karatsuba. Nous

avons conçu plusieurs algorithmes pour ces optimisations et avons évalué leur complexité dans le cas de codes écrits en langage C. Nous avons implanté ces approches et présenté les résultats obtenus sur deux plates-formes différentes : Intel Core 2 et Intel Core i5.

Nous avons aussi implanté la multiplication scalaire avec des opérations de points sur courbes $E(\mathbb{F}_{2^{233}})$ et $E(\mathbb{F}_{2^{409}})$ incluant ces optimisations, tout en utilisant les deux optimisations appelées *Lazy-reduction*. Les résultats expérimentaux ont montré que l'optimisation $AB + CD$ apporte une amélioration des performances sur Intel Core 2 seulement pour le corps $\mathbb{F}_{2^{233}}$. Sur Intel Core i5, cette optimisation apporte les meilleurs résultats pour la multiplication scalaire pour les deux corps $\mathbb{F}_{2^{233}}$ et $\mathbb{F}_{2^{409}}$. Sur cette dernière plate-forme, nous avons atteint les meilleurs résultats de performances publiés dans la littérature, en particulier dans [64].

Chapitre 6

Paralléliser l'échelle binaire de Montgomery

Les processeurs modernes sont constitués de plusieurs cœurs. En effet, il y a en général deux ou quatre cœurs physiques dans les processeurs généralistes que l'on trouve dans nos ordinateurs de bureaux ou portables, tels que ceux produits par Intel ou AMD, mais aussi dans la plupart des plates-formes mobiles (téléphones ou tablettes). Ce nombre de cœurs va probablement croître dans le futur. Tirer avantage de cette multiplication des cœurs est un défi à relever qui se formule de la façon suivante : comment améliorer les performances et/ou la résistance aux attaques en parallélisant les implantations logicielles ? Dans le domaine de la cryptographie, il est assez naturel d'effectuer plusieurs tâches en parallèle, par exemple, le chiffrement de plusieurs messages. En revanche, le défi à relever est plus difficile en ce qui concerne les opérations internes aux protocoles, la multiplication scalaire de points de courbe elliptique en particulier. Il semble intéressant de paralléliser ces calculs si cela permet de réduire la latence de ces opérations.

Dans le cas de la caractéristique 2, nous avons vu que nous pouvions diviser un point de courbe elliptique par deux efficacement, et que ceci permet d'élaborer des algorithmes de multiplication scalaire plus efficaces que les algorithmes classiques *Double-and-add*, les algorithmes *Halve-and-add* et parallèles *Double/halve-and-add* (les algorithmes 2.9 page 71 et 2.10 page 72).

Dans ce chapitre, nous nous intéressons à l'application de cette technique de parallélisation à l'échelle binaire de Montgomery sur courbe $E(\mathbb{F}_{2^m})$. Nous avons présenté l'échelle binaire de Montgomery en section 3.2.2.3, l'algorithme 3.6 page 85. Cet algorithme de multiplication scalaire présente une grande régularité dans la séquence des opérations et constitue donc une bonne parade à l'attaque SPA comme nous l'avons montré en section 3.2.2. Malheureusement, sa complexité reste plus élevée que les algorithmes séquentiels les plus performants, comme le montre le comparatif que nous avons fourni dans la table 2.13 page 73.

Nos contributions sont les suivantes :

— élaboration d'un algorithme d'échelle binaire de Montgomery basé sur la *halving* de point ;
— élaboration d'un algorithme parallèle dans l'esprit de l'algorithme parallèle *Double/halve-and-add* utilisant les deux échelles binaires de Montgomery, l'une basée sur le doublement et l'autre basée sur le *halving* ;
— implantation de ces algorithmes sur deux plates-formes, dont une mobile. Nous sélectionnons la meilleure configuration de parallélisation et évaluons les gains de performances.

Ces contributions font partie d'un article publié dans la revue *IEEE - Transactions on Computers*, 2015 [52].

6.1 Échelle binaire de Montgomery à base de *halving* et parallèle

6.1.1 *Montgomery-halving*

Nous considérons une courbe elliptique ordinaire (non super-singulière) $E(\mathbb{F}_{2^m})$ definie par son équation courte $y^2 + xy = x^3 + ax^2 + b$ sur $\mathbb{F}_{2^m}$, un point $P \in E(\mathbb{F}_{2^m})$ d'ordre impair r et un scalaire $k \in [0, r-1]$ de longueur t bits. L'ordre impair du point P permet d'utiliser le *halving* dans le sous-groupe engendré par P, comme nous l'avons vu en section 2.3.3.2.

Notre but est de proposer une nouvelle version de l'échelle binaire de Montgomery utilisant la *halving* de points à la place du doublement. Cet algorithme aura les propriétés suivantes, comme l'échelle binaire de Montgomery classique :

— les opérations effectuées durant chaque itération de la boucle `for` principale seront régulières afin de préserver la propriété de résistance à l'attaque SPA ;

— la différence entre les deux points Q_1 et Q_0 sera constante et le résultat final dépendra de toutes les opérations effectuées, ceci apportant la résistance aux attaques par injection de fautes et de type *safe error*.

À l'instar de l'approche *Halve-and-add* (voir sections 2.3.3.2 et 2.3.4.4), nous récrivons le scalaire k en calculant au préalable

$$k' = 2^{t-1} \cdot k \bmod r = \sum_{i=0}^{t-1} k'_i 2^i.$$

Nous obtenons donc $k = \sum_{i=0}^{t-1} k'_i 2^{i-(t-1)}$ où $k'_i \in \{0, 1\}$ et t est la longueur binaire de r. Nous utilisons aussi deux points Q_0 et Q_1 comme dans l'échelle binaire de Montgomery classique.

Notre proposition est la suivante :

— le point Q_0 prend successivement les valeurs $k^{(i)} \cdot P$ pour $i = 0, 1, \ldots, t-1$ où $k^{(i)} = \sum_{j=0}^{i} k'_j 2^{j-i}$;

— le point Q_1 satisfait quant à lui $Q_1 = Q_0 - 2P$ durant toute l'exécution de l'algorithme.

Notre proposition de multiplication scalaire *Montgomery-halving* est décrite dans l'algorithme 6.1.

Algorithme 6.1 *Montgomery-halving*

Require: $P \in E(\mathbb{F}_{2^m})$ d'ordre impair r et un scalaire $k \in [0, r-1]$, et $t = \lceil \log_2(r) \rceil$.
Ensure: $Q = k \cdot P$

1: Calcul préalable de $k' = 2^{t-1} \cdot k \bmod r = \sum_{i=0}^{t-1} k'_i 2^i$
2: $Q_0 \leftarrow k'_0 \cdot P, Q_1 \leftarrow k'_0 \cdot P - 2 \cdot P$
3: **for** $i = 1$ to $t-1$ **do**
4: **if** $(k'_i = 1)$ **then**
5: $T \leftarrow Q_0/2, Q_0 \leftarrow T, Q_1 \leftarrow Q_1 - T$
6: **else**
7: $T \leftarrow Q_1/2, Q_1 \leftarrow T, Q_0 \leftarrow Q_0 - T$
8: **end if**
9: **end for**
10: **return** (Q_0)

Le lemme suivant prouve la validité du fonctionnement de cet algorithme.

Lemme 6.1.1. *Soient $Q_{0,i}$ et $Q_{1,i}$ les valeurs respectives de Q_0 et Q_1 à l'issue de la $i^{ème}$ itération de la*

boucle et soit $k^{(i)} = \sum_{j=0}^{i} k'_j 2^{j-i}$. *Alors, les identités suivantes sont vérifiées :*

$$\begin{cases} Q_{0,i} &= k^{(i)} \cdot P, \\ Q_{1,i} &= (k^{(i)} - 2) \cdot P. \end{cases}$$

Démonstration. Nous démontrons ce lemme par récurrence sur i. En premier lieu, examinons le cas $i = 0$: l'initialisation de Q_0 et Q_1 donne respectivement $Q_0 = k'_0 \cdot P$ et $Q_1 = (k'_0 - 2) \cdot P$. Ceci vérifie bien l'hypothèse de récurrence. On suppose maintenant que l'hypothèse de récurrence est vérifiée pour i. Nous allons démontrer qu'elle est aussi vérifiée pour $i+1$. Pour l'itération i, par notre hypothèse de récurrence, nous avons :

$$\begin{cases} Q_{0,i} &= k^{(i)} \cdot P, \\ Q_{1,i} &= k^{(i)} \cdot P - 2 \cdot P, \end{cases} \quad \text{et } Q_{1,i} - Q_{0,i} = -2 \cdot P.$$

Nous examinons chacun des deux cas possibles :

— Si $k'_{i+1} = 0$, c'est à dire $k^{(i+1)} = k^{(i)}/2$, alors, l'algorithme exécute les opérations suivantes :

$$\begin{cases} Q_{0,(i+1)} &= Q_{0,i}/2 = (k^{(i)}/2) \cdot P, \\ Q_{1,(i+1)} &= Q_{1,i} - Q_{0,i}/2 = (Q_{1,i} - Q_{0,i}) + Q_{0,i}/2. \end{cases}$$

Par l'hypothèse de récurrence, $Q_{1,i} - Q_{0,i} = -2P$ et $Q_{0,i} = k^{(i)}P$ ce qui implique $Q_{0,i}/2 = (k^{(i)}/2) \cdot P = k^{(i+1)}P$. En conséquence, les valeurs suivantes sont obtenues :

$$\begin{cases} Q_{0,(i+1)} &= k^{(i+1)} \cdot P, \\ Q_{1,(i+1)} &= -2 \cdot P + k^{(i+1)}P. \end{cases}$$

— Si $k'_{i+1} = 1$, c'est à dire $k^{(i+1)} = k^{(i)}/2 + 1$, les deux points $Q_{0,i+1}$ et $Q_{1,i+1}$ sont calculés ainsi :

$$\begin{cases} Q_{0,(i+1)} &= Q_{0,i} - Q_{1,i}/2 = Q_{0,i}/2 - (Q_{1,i} - Q_{0,i})/2, \\ Q_{1,(i+1)} &= Q_{1,i}/2 \end{cases}$$

À nouveau, l'hypothèse de récurrence fournit $Q_{1,i} - Q_{0,i} = -2P, Q_{0,i} = k^{(i)}P$ et $Q_{1,i} = (k^{(i)} - 2)P$. On en déduit

$$\begin{cases} Q_{0,(i+1)} &= (k^{(i)}/2)P + P = k^{(i+1)}P, \\ Q_{1,(i+1)} &= \frac{(k^{(i)}-2)}{2}P = (k^{(i)}/2 + 1)P - 2P = k^{(i+1)}P - 2P. \end{cases}$$

Ceci termine la démonstration du lemme. □

En terme de complexité pour l'algorithme 6.1, on a un doublement (pour calculer $2 \cdot P$) et une addition de points dans l'étape 2, puis, dans les étapes 3 à 9 une boucle principale de $t - 1$ itérations comportant chacune un *halving* et une addition de points. On remarque que les additions de points sont en coordonnées affines. Ceci se traduit dans la table suivante :

Algorithme	Coord.	Opérations sur points de courbe		
		Additions	*Halvings*	Doublements
Montgomery-halving Algo. 6.1	affines	t	$t-1$	1

L'algorithme proposé présente une régularité dans les opérations réalisées à chaque itération de la boucle principale, c'est à dire que la séquence d'opérations est indépendante de la valeur du bit k'_i. La différence $Q_{1,i} - Q_{0,i} = -2 \cdot P$ est maintenue tout au long du déroulement de l'algorithme, ce qui peut être utilisé pour effectuer une détection d'injection d'erreurs. Par ailleurs, aucun calcul n'est inutile, ce qui garantit aussi une résistance à l'attaque *safe error*.

Ainsi, l'algorithme proposé conserve les propriétés principales de l'échelle binaire de Montgomery originale.

Hélas, en terme de complexité, l'utilisation du *halving* de point oblige à recourir à une addition en coordonnées affines dans la boucle principale de l'algorithme. Nous n'avons en effet pas pu construire une version en coordonnées projectives de notre approche *Montgomery-halving*, en vue d'économiser l'inversion qui en résulte dans les t additions de points de notre algorithme. En revanche, le développement d'une version parallèle de l'échelle binaire de Montgomery reste possible. La section suivante présente cette approche.

6.1.2 Approche parallèle

Dans cette version parallèle, nous allons utiliser la technique de section du scalaire en deux parties proposées par Taverne *et al.* dans [64]. Soit P le point d'ordre impair r que l'on souhaite multiplier par un scalaire k. La technique de parallélisation est celle que nous avons présentée en section 2.3.4.4. Pour ce faire, nous calculons

$$k' = 2^{\ell} \times k \bmod r = \sum_{i=0}^{\ell-1} k'_i 2^i$$

ce qui produit une décomposition en deux parties $k = k_1 + k_2$, où k_1 contient des puissances positives de 2 et k_2 des puissances négatives de 2 comme suit :

$$k = k' \times 2^{-\ell} \bmod r = \underbrace{(k'_{\ell-1}2^{\ell-1-n} + \ldots + k'_n)}_{k_1} + \underbrace{(k'_{n-1}2^{-1} + \ldots + k'_0 2^{-n})}_{k_2} \bmod r.$$

La multiplication scalaire $k_1 \cdot P$ est alors effectuée avec l'échelle binaire de Montgomery originale et la deuxième partie de la multiplication scalaire $k_2 \cdot P$ est effectuée en parallèle par l'approche *Montgomery-halving*. Le résultat final est obtenu à l'aide d'une addition finale $k \cdot P = (k_1 \cdot P) + (k_2 \cdot P)$.

6.2 Implantations logicielles

Nous avons réalisé ces implantations sur deux plates-formes différentes. Une implantation classique sur ordinateur de bureau, et une autre sur plate-forme mobile. En effet, la résistance aux attaques de type SPA est un enjeu plus important sur ce dernier type d'appareil dont l'accès physique est par définition plus facile, et dont la vulnérabilité est aussi plus grande comme nous l'avons vu en section 3.1. Voici les principales caractéristiques de ces deux plates-formes :

— L'ordinateur de bureau Optiplex 990 DELL muni d'un processeur Intel Core i7-2600. Le système d'exploitation est la distribution Linux Ubuntu 12.04. Le code, écrit en langage C, a été compilé à l'aide de gcc 4.6.3. Suivant les recommandations de [8], les options du processeur *Hyperthreading* et *Turbo-boost* ont été désactivées afin de mesurer les performances de nos algorithmes de la façon la plus précise.

— La tablette Google Nexus 7 équipée d'un processeur Qualcomm Snapdragon (architecture bi-cœurs ARM-v7). Le système d'exploitation est aussi Linux, sous la distribution mobile Ubuntu touch 10.04. Le code, écrit en langage C, a été compilé à l'aide de gcc 4.8.2. La bibliothèque Papi [1] a été utilisée pour obtenir le comptage des cycles d'horloge du processeur, afin de mesurer les performances.

Nous avons implanté la multiplication scalaire sur deux courbes recommandées par le NIST, la courbe B233 définie par $\mathbb{F}_{2^{233}} = \mathbb{F}_2[t]/(t^{233} + t^{87} + 1)$ et la courbe B409 définie par

$\mathbb{F}_{2^{409}} = \mathbb{F}_2[t]/(t^{409} + t^{79} + 1)$. Ces deux courbes sont données par leur équation courte de Weierstrass $y^2 + xy = x^3 + x^2 + b$ où b est un élément non creux du corps $\mathbb{F}_{2^m}$. Ces courbes remplissent les conditions requises pour un *halving* de point efficace : $a = 1$ d'où $\mathrm{Tr}(a) = 1$, et un point $P \in E(\mathbb{F}_{2^m})$ d'ordre impair r (voir la section 2.3.3.2).

6.2.1 Stratégies d'implantation des opérations sur les corps $\mathbb{F}_{2^{233}}$ et $\mathbb{F}_{2^{409}}$

Les stratégies d'implantation des opérations sur le corps sont souvent similaires à celles présentées au chapitre précédent, en particulier dans la section 5.4.2. Néanmoins, certaines différences sont à mentionner sur certains points et qui concernent principalement le cas de l'implantation sur la Nexus 7 équipée du Qualcomm Snapdragon, du fait de l'architecture 32 bits de ce processeur. Nous insistons donc essentiellement sur ces spécificités. Quand nous ne mentionnons rien, c'est que la démarche est strictement celle appliquée en section 5.4.2 et le lecteur peut s'y reporter si besoin.

— *Multiplication des polynômes binaires :* Sur processeur Snapdragon, l'architecture ARM ne fournit pas d'instruction de multiplication sans retenue. Nous utilisons donc l'approche de Lopez et Dahab présentée dans l'algorithme 1.12 page 40, mais adaptée à la taille des mots de cette architecture, qui est de 32 bits seulement.

— *Élévation au carré :* Sur processeur Snapdragon, nous appliquons la technique d'insertion des zéros par l'utilisation de table que nous avons décrite dans l'algorithme 1.14 page 43, en adaptant simplement à la taille de mots de 32 bits de cette architecture.

— *Opérations modulo r :* Sur les deux plates-formes, nous avons implanté la multiplication multiprécision avec l'approche *schoolbook* et la réduction modulo r en utilisant l'approche de Montgomery [46]. Pour l'inversion, nous avons utilisé la fonction de bas niveau pour l'algorithme d'Euclide étendu de la bibliothèque `GMP` dans [2].

6.2.2 Performances des implantations logicielles

Nos implantations ont été réalisées pour deux courbes elliptiques sur corps binaires préconisées par le NIST [19], les courbes B233 et B409 dont les paramètres figurent dans l'annexe 8.1.1.

Dans la table 6.1, nous transcrivons les complexités en nombre d'instructions (WXOR, WAND et décalages) et les tailles des tables requises ainsi que les nombres de cycles pour les opérations élémentaires sur les corps finis.

On remarque que sur nos deux plates-formes Core i7 et Snapdragon, les opérations les plus rapides sont le calcul du carré et de la racine carrée. On remarque de plus que les opérations de multiplication, *Half-Trace* et *multisquaring* ont approximativement le même coût. L'inversion est l'opération la plus longue : de 22 à 30 multiplications sur Intel Core i7, et 16 à 18 multiplications sur Snapdragon.

Dans la table 6.2, nous transcrivons les performances obtenues pour les trois multiplications scalaires considérées : l'approche classique de Montgomery (l'algorithme 3.6 page 85), l'approche *Montgomery-halving* (l'algorithme 6.1 page 128) et l'approche parallèle présentée dans la section 6.1.2. Nous fournissons également les résultats des approches *Double-and-add*, *Halve-and-add* et *Parallel Double/halve-and-add*, qui sont les plus performantes de l'état de l'art, mais ne sont pas protégées contre l'attaque SPA.

En complément, nous avons aussi simulé un scénario de signature selon le protocole ECDSA qui est également susceptible d'être vulnérable aux attaques par canal auxiliaire. La signature

TABLE 6.1 – Complexité et performance des opérations sur $\mathbb{F}_{2^m}$.

Intel Core i7								
Opération sur le corps	$\mathbb{F}_{2^{233}}$				$\mathbb{F}_{2^{409}}$			
	#CC	ns	Nb Op.	taille table (octets)	#CC	ns	Nb Op.	taille table (octets)
Multiplication	98	29	69	-	258	76	191	-
Carré	17	5	36	-	34	10	73	-
Racine carrée	52	15	104	-	48	14	72	-
Inversion	2162	635	2265	90624	7317	2152	5235	421888
Half-Trace	130	38	177	30208	255	75	309	105472

Opération mod r	$E(\mathbb{F}_{2^{233}})$		$E(\mathbb{F}_{2^{409}})$	
	#CC	ns	#CC	ns
Multiplication	184	54	726	213
Inversion avec [2]	3547	1043	6046	1778

Qualcomm Snapdragon								
Opération sur le corps	$\mathbb{F}_{2^{233}}$				$\mathbb{F}_{2^{409}}$			
	#CC	ns	Nb Op.	taille table (octets)	#CC	ns	Nb Op.	taille table (octets)
Multiplication	2071	1380	851	-	5995	3996	1897	-
Carré	137	91	151	1024	229	152	187	1024
Racine carrée	275	180	356	1024	368	245	412	1024
Inversion	33416	22277	14659	84960	110090	73393	31519	342784
Half-Trace	1735	1156	590	28320	5753	3835	1545	85696

Opération mod r	$E(\mathbb{F}_{2^{233}})$		$E(\mathbb{F}_{2^{409}})$	
	#CC	ns	#CC	ns
Multiplication	1375	916	4611	3074
Inversion avec [2]	8573	5715	16328	3201

d'un message $\mathcal{M}$, ou un haché de celui-ci, consiste en un couple (s, R_x) calculé comme suit :

$$\begin{aligned} k &\leftarrow Random(1, r), \\ R = (R_x, R_y) &\leftarrow k \cdot P, \\ s &\leftarrow k^{-1}(\mathcal{M} + R_x \times \mathcal{C}_{pr}) \bmod r. \end{aligned}$$

où r est l'ordre du point P et $\mathcal{C}_{pr}$ la clé privée. Dans notre implantation parallèle, l'inversion modulaire $k^{-1} \bmod r$ est effectuée par l'un des deux *threads* de la multiplication scalaire. Les autres opérations modulo r pour obtenir la valeur de s, nécessitent de connaître R et sont donc effectuées après la jointure des deux *threads*. Elles sont donc exécutées séquentiellement en fin de calcul.

Les performances reportées dans la table 6.2 sont mesurées comme suit : les points P et les scalaires k sont générés aléatoirement et les valeurs de chaque temps d'exécution sont moyennées sur plusieurs centaines de jeux de données. La valeur indiquée dans la colonne intitulée *split* correspond à la taille ℓ du scalaire partiel k_2, c'est à dire le nombre d'itérations de la boucle principale du *thread* effectuant l'approche *Montgomery-halving*, tel que défini dans la section 6.1.2, et qui minimise le temps de calcul global de l'approche parallèle. La valeur optimale de ℓ est donnée dans la table suivante pour chaque cas :

Courbe elliptique	B233	B409
Intel Core i7	*split* = 40	*split* = 60
Qualcomm Snapdragon	*split* = 50	*split* = 85

La figure 6.1 représente le comportement de l'approche parallèle quand la valeur du *split* ℓ varie de 20 à 60 pour la courbe B233. On remarque que la courbe à une forme en «V» caractéristique. Chaque branche du «V» peut être approchée par une droite et les performances

TABLE 6.2 – Performances des multiplications scalaires et de la signature ECDSA dans $E(\mathbb{F}_{2^m})$, sur processeurs Intel Core i7 et Qualcomm SnapDragon.

Intel Core i7								
Algorithme	#Core	Operat.	B233			B409		
			#CC/10^3	μs	split	#CC/10^3	μs	split
γ-NAF Double-and-Add avec $\gamma = 4$	1	SM	147	43	-	654	192	-
		ECDSA	152	44	-	666	196	-
γ-NAF Halve-and-Add avec $\gamma = 4$	1	SM	129	38	-	521	153	-
		ECDSA	134	39	-	542	159	-
γ-NAF Parallel Double/halve-and-add avec $\gamma = 4$	2	SM	97	29	134	337	99	234
		ECDSA	104	31	128	349	102	231
Montgomery-Ladder	1	SM	149	44	-	694	204	-
		ECDSA	154	45	-	710	208	-
Montgomery-Halving	1	SM	689	202	-	3826	1125	-
		ECDSA	691	203	-	3869	1137	-
Montgomery-Parallel	2	SM	143	42	39	624	184	59
		ECDSA	147	43	40	641	188	64
Gain relatif Mont-Par./Mont-Ladder			5.09 %			10.5 %		

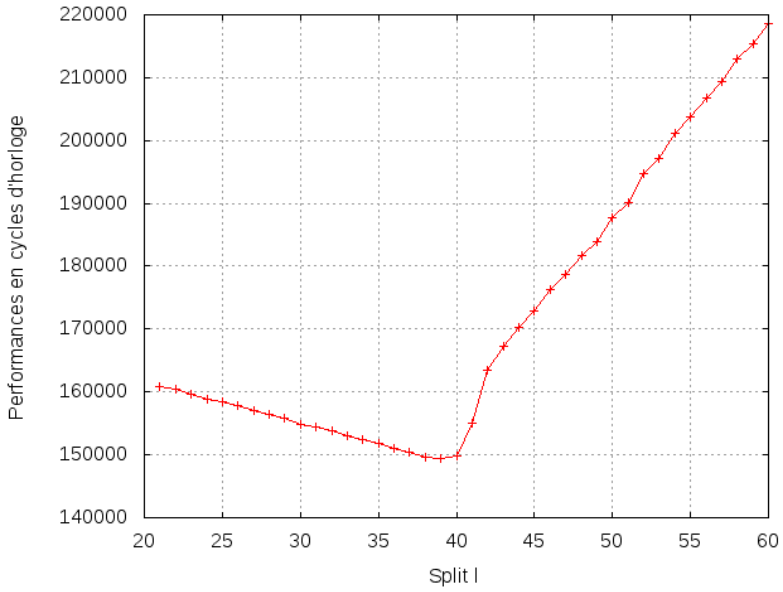
Qualcomm Snapdragon								
Algorithme	#Core	Operat.	B233			B409		
			#CC/10^3	μs	split	#CC/10^3	μs	split
γ-NAF Double-and-Add avec $\gamma = 4$	1	SM	3114	2065	-	14747	9831	-
		ECDSA	3118	2079	-	14839	9892	-
γ-NAF Halve-and-Add avec $\gamma = 4$	1	SM	2373	1568	-	11401	7601	-
		ECDSA	2456	1.63	-	11438	7625	-
γ-NAF Parallel Double/halve-and-add avec $\gamma = 4$	2	SM	1591	1060	127	6395	4263	246
		ECDSA	1607	1071	127	6455	4303	246
Montgomery-Ladder	1	SM	3407	2258	-	16311	10874	-
		ECDSA	3434	2289	-	16440	10960	-
Montgomery-Halving	1	SM	11337	7528	-	59810	39873	-
		ECDSA	11687	7791	-	59847	39898	-
Montgomery-Parallel	2	SM	2756	1824	56	13155	8770	86
		ECDSA	2759	1839	51	13206	8803	88
Gain relatif Mont-Par./Mont-Ladder			19 %			19 %		

peuvent être ainsi prédites par une équation en fonction de ℓ. Nous donnons ici l'équation correspondante à l'approche parallèle appliquée à la courbe elliptique B233 :

$$\#cycles \approx \max(-607\ell + 164968, 3022\ell + 31018)$$

Sur les performances obtenues par notre algorithme parallèle utilisant l'échelle binaire de Montgomery, nous remarquons un gain de l'ordre de 5 à 10 % en comparaison de l'échelle binaire de Montgomery classique sur Intel Core i7. L'accélération est de 19 % considérant les mêmes approches sur Qualcomm Snapdragon. Le gain supérieur sur cette dernière plate-forme s'explique en partie par le ratio de coût entre l'inversion et la multiplication sur les corps $\mathbb{F}_{2^{233}}$ et $\mathbb{F}_{2^{409}}$, qui est plus petit et donc plus favorable dans ce cas. En effet ceci se répercute sur les performances relatives des approches *Montgomery-halving* et Montgomery classique utilisées dans les deux *threads*. Les valeurs de *split* ℓ sont alors légèrement plus importantes.

FIGURE 6.1 – Temps d'exécution des échelles de Montgomery parallèles en fonction du split ℓ



6.3 Conclusion sur l'échelle binaire de Montgomery parallélisée

En complément de nos résultats expérimentaux, nous fournissons ci-après quelques résultats de références publiés dans la littérature pour quelques approches concernant la multiplication scalaire par l'échelle binaire de Montgomery, pour des tailles de corps proches de 256. Les résultats d'implantation sont tous obtenus sur plate-forme Intel Core i7 Sandy Bridge ou i5 Ivy Bridge. Ils montrent que nos propres implantations sont compétitives du point de vue de l'état de l'art, à ceci près que la comparaison nous est favorable du fait de la taille du corps $\mathbb{F}_{2^{233}}$ que nous avons utilisé, avec un niveau de sécurité de 112 bits et qui est inférieur à celui des auteurs dont nous présentons les résultats. Cependant, ces implantations tirent parti de coefficients de courbe creux.

	courbe	type de coeff. pour la courbe	niveau de sécurité	processeur	#core	#cycles
Taverne *et al.* [64]	Curve2251	creux	125	SB	1	225 000
Bernstein [8]	Curve25519	creux	128	SB	1	194 000
Hamburg [21]	$E(\mathbb{F}_{p_{253}})$	creux	126	SB	1	153 000

Nous remarquons tout de même que nos performances sur les approches suivantes : l'échelle binaire de Montgomery, *Double-and-add*, *Halve-and-add* et *Parallel Double/halve-and-add* reportées dans la table 6.2 sont meilleures que celles publiées par Taverne *et al.* dans [64].

Concernant le processeur Qualcomm Snapdragon, on trouve dans [8] des performances d'implantations logicielles pour des courbes elliptiques de niveau de sécurité 128 comprises entre 410106 et 15472248 cycles d'horloge. Malheureusement, aucun de ces résultats ne correspond à des courbes du NIST. Le coût élevé des opérations sur le corps binaire dans le cas de cette plate-forme explique le fossé de performance en comparaison de multiplications scalaires sur courbe $E(\mathbb{F}_p)$, i.e., 410106 cycles, et nos performances pour les courbes $E(\mathbb{F}_{2^m})$ présentées dans la table 6.2.

Nous pensons que des implantations matérielles (i.e. ASIC ou FPGA) peuvent tirer un bénéfice encore plus intéressant de notre algorithme parallèle. En effet, l'usage d'une représentations des éléments du corps en base normale dans le cas d'implantation matérielle est de nature à améliorer l'efficacité de l'inversion relativement à la multiplication, du fait de la quasi gratuité de l'opération d'élévations multiples au carré. De ce fait, l'approche *Montgomery-halving* verra ses performances se rapprocher de celles de l'échelle binaire de Montgomery classique, et par là, augmenter significativement le bénéfice de l'approche parallèle.

Pour finir, on relève que notre proposition est résistante à l'attaque SPA, et aussi à la *Timing Attack* dans les hypothèses formulées au chapitre 3. L'objectif d'améliorer les performances en préservant cette propriété est bien atteint.

Chapitre 7

Protection par la parallélisation

Dans le précédent chapitre, nous avons traité de l'adaptation de l'approche parallèle proposée par Taverne *et al.* à l'échelle binaire de Montgomery pour la multiplication scalaire de point de courbe elliptique sur corps binaire. Nous abordons maintenant une approche différente, l'implantation logicielle d'algorithmes parallèles de multiplication scalaire de point de courbe elliptique de type *Double-and-add* ou *Halve-and-add* (en caractéristique 2) avec réécriture du scalaire en représentation NAF ou γ-NAF (voir section 2.3.4). Notre objectif est ici d'améliorer les performances.

Moreno et Hasan dans [48] ont proposé une approche régulière pour la multiplication scalaire basée sur l'algorithme *Right-to-left Double-and-add* résistante à l'attaque SPA. Cette approche utilise une technique de *buffer* et la version parallèle améliore les performances. Cependant, ils n'ont pas réalisé d'implantations, leurs travaux se sont en effet limités à une étude théorique. C'est ce que nous avons réalisé pour leur approche parallèle et que nous présentons dans ce chapitre. Nous avons en plus étendu cette approche en la combinant avec la technique de parallélisation de Taverne *et al.* [65] résultant en un algorithme de multiplication scalaire sur courbe sur corps binaire à quatre *threads*. Ce travail a été présenté à Beijing le 15 décembre 2014, lors de la conférence INSCRYPT, et publié dans les actes de cette conférence [55].

Les implantations logicielles proposées tiennent compte des aspects suivants :

— Nous analysons trois stratégies différentes pour les mécanismes de synchronisation entre *threads* : utilisation de `signals`, `mutexes`, ou encore l'approche `busy-waiting`. Nous proposons un mécanisme de synchronisation issu de cette analyse.

— Nous étudions aussi l'impact de la représentation du scalaire k, auquel est lié le nombre d'additions et de calculs de reconstruction finale, c'est à dire la charge de calcul du *thread* effectuant les additions.

— Nous proposons aussi de combiner l'approche parallèle *Double/halve-and-add* de Taverne *et al.* [65] (présentée dans la section 2.3.4.4) avec l'approche inspirée de Moreno et Hasan. Ceci nous permet de proposer un algorithme de multiplication scalaire de point de courbe elliptique sur $E(\mathbb{F}_{2^m})$ à quatre *threads*.

— Nous fournissons des résultats expérimentaux pour deux courbes elliptiques sur corps premier $\mathbb{F}_p$ avec $p = 2^{255} - 19$ et pour deux courbes elliptiques sur corps binaires préconisées par le NIST dans [19], B233 et B409. Ces résultats expérimentaux montrent que l'accélération des calculs de multiplication scalaire va jusqu'à 15 % en comparaison des homologues séquentiels.

Ce chapitre est organisé de la manière suivante : en section 7.1, après avoir rappelé quelques généralités sur la parallélisation, nous présentons la démarche de Moreno et Hasan dans [48] ; en section 7.2, nous présentons les algorithmes parallèles, leur implantation et leur performance ; nous terminons par quelques remarques de conclusion.

7.1 Parallélisation, algorithme de Moreno et Hasan [48], résistance à l'attaque SPA

7.1.1 Généralités sur la parallélisation de l'approche *Double-and-add*

En guise de préliminaire, on rappelle les boucles principales des deux algorithmes génériques (extrait des algorithmes 2.4 et 2.5 page 68) :

Left-to-right Double-and-add Algorithme 2.4	*Right-to left Double-and-add* Algorithme 2.5
1: $Q \leftarrow \mathcal{O}$ 2: **for** $i = t-1$ downto 0 **do** 3: $\quad Q \leftarrow 2 \cdot Q$ 4: $\quad$ **if** $k_i = 1$ **then** 5: $\quad\quad Q \leftarrow Q + P$ 6: $\quad$ **end if** 7: **end for**	1: $Q \leftarrow \mathcal{O}$ 2: **for** $i = 0$ to $t-1$ **do** 3: $\quad$ **if** $k_i = 1$ **then** 4: $\quad\quad Q \leftarrow Q + P$ 5: $\quad$ **end if** 6: $\quad P \leftarrow 2 \cdot P$ 7: **end for**

Dans le cas de l'algorithme 2.4 *Left-to-right Double-and-add*, on remarque une dépendance lecture-après-écriture : l'étape 3 d'une itération de la boucle principale ne peut être effectuée indépendamment de l'étape 5 de l'itération précédente. Cet algorithme est donc fortement séquentiel et ne peut être parallélisé en l'état.

En revanche, l'algorithme 2.5 *Right-to-left Double-and-add* est différent dans son principe. En effet, dans ce cas, on évalue l'expression suivante :

$$k \cdot P = (\sum_{i=0}^{t-1} 2^i k_i) \cdot P = \sum_{i=0}^{t-1} k_i (2^i \cdot P).$$

Le *Halve-and-add* (l'algorithme 2.9 page 71) se comporte de la même manière. Après réécriture du scalaire $k' = k \cdot 2^{t-1} \bmod r$ où r est l'ordre impair du point P, il évalue en effet

$$k \cdot P = (\sum_{i=0}^{t-1} k'_i 2^{i-(t-1)}) \cdot P = \sum_{i=0}^{t-1} k'_i (2^{i-(t-1)} \cdot P).$$

Revenons au cas *Right-to-left Double-and-add*. Deux niveaux de parallélisme se dégagent :

— chaque terme $(2^i P)$ constitue un atome, et la somme peut être évaluée en utilisant des techniques de parallélisation une fois que chacun des atomes est connu ;

— d'autre part, l'évaluation de chaque atome $(2^i P)$ peut-être faite indépendamment et de façon concurrente de la somme, en respectant les dépendances lecture-après-écriture.

Dans ce chapitre, nous nous intéressons à la deuxième voie. Dans la suite, nous implanterons des algorithmes qui calculent chaque doublement (ou *halving* de points le cas échéant) à l'aide d'un processeur ou cœur, et la somme des termes en parallèle sur un deuxième cœur.

On note pour finir que ce que nous venons d'évoquer s'applique aussi aux exponentiations modulaires présentées dans les algorithmes 2.1 et 2.2 page 53.

7.1.2 Présentation de l'algorithme parallèle de Moreno et Hasan [48]

Moreno et Hasan dans [48] proposent différents algorithmes dans le but d'apporter de la résistance à l'attaque SPA présentée en section 3.2. Ils se placent dans le cas générique d'exponentiation ou multiplication scalaire de point de courbe elliptique, leur idée s'appliquant

indifféremment dans l'un ou l'autre cas. Nous décrivons leur démarche dans la formulation correspondant au cas de la multiplication scalaire.

Partant de l'algorithme *Right-to-left Double-and-add*, le principe est de mémoriser un certain nombre de doublements de points relatifs aux chiffres binaires du scalaire non nuls dans un espace mémoire tampon et de les utiliser ensuite en les additionnant. Nous reproduisons ici leur approche parallèle, l'algorithme 7.1 dans lequel on note $\mathcal{F}$ l'inverse de la proportion de chiffres non nuls de la représentation du scalaire.

Algorithme 7.1 *Double-and-add* parallèle Moreno et Hasan [48]

Require: $k = (k_{t-1}, \ldots, k_1, k_0), P \in E(\mathbb{F}_q)$.
Ensure: kP
(Barrier)

thread 1 (doublements)
1: $Q \leftarrow P$
2: **for** $i = 1$ to $t - 1$ **do**
3: **if** $k_i = 1$ **then**
4: $P \rightarrow$ *buffer*$_Q$
5: **end if**
6: $P \leftarrow 2 \cdot P$
7: **if** $i \bmod \mathcal{F} = 0$ **then**
8: Signal *thread* 2
9: **end if**
10: **end for**

thread 2 (additions)
11: **if** signal du *thread* 1 **then**
12: $T \leftarrow$ *buffer*$_Q$
13: $Q \leftarrow Q + T$
14: **end if**

(Barrier)
15: **return** Q

On remarque que le signal étape 8 est envoyé tous les $\mathcal{F}$ tours de boucle, selon le type de représentation du scalaire (binaire, signée ou non, γ-NAF...) Ainsi, la mémoire tampon, appelée *buffer* dans l'algorithme, se vide de façon constante : tous les $\mathcal{F}$ tours de boucle, pour $i \bmod \mathcal{F} = 0$, on consomme une valeur de P. La proportion attendue de chiffres non nuls de $1/\mathcal{F}$ justifie de vider le *buffer* à ce rythme. Par exemple, dans le cas d'une représentation NAF (voir la section 2.3.4.2), la proportion de chiffres non nuls est de $1/3$, et dans ce cas, on choisit $\mathcal{F} = 3$.

En revanche, le *buffer* se remplit au rythme des bits k_i non nuls. Il peut donc se produire deux phénomènes :

— le *buffer* se remplit plus vite qu'il ne se vide et si sa taille est insuffisante, on a débordement ;
— le *buffer* se vide trop vite et on a alors tarissement.

Moreno et Hasan traitent ces deux questions dans leur article et nous n'en dirons pas plus sur le sujet. Nous nous proposons d'implanter leur algorithme parallèle, et c'est ce que nous présentons dans la suite.

Robustesse face aux attaques SPA et *Timing*

Pour tous leurs algorithmes, Moreno et Hasan revendiquent une résistance à l'attaque SPA à cause de la régularité des opérations de doublements et additions effectuées. Ils préconisent tout de même de prévoir des opérations d'écriture en *buffer* factices dans le cas de chiffres non nuls si ces opérations sont visibles sur une trace SPA.

Ils remarquent néanmoins un effet collatéral de leur algorithme. En effet, contrairement aux algorithmes réguliers que nous avons présentés dans la section 3.2.2, le temps d'exécution du calcul n'est pas indépendant du scalaire. La situation est donc celle que nous avons évoquée en début de section 3.3 : un attaquant peut connaître le nombre de chiffres non nuls du scalaire par chronométrage, avec l'avantage que cela procure.

Nous notons toutefois que ceci ne constitue pas à proprement parler une faiblesse face à une *Timing attack* qu'un attaquant peut monter selon la dépendance du temps d'exécution de l'addition ou du doublement en fonction des données d'entrées. On se reportera à la section 3.3 pour la présentation de cette attaque.

7.2 Algorithmes parallèles et implantations logicielles

Notre objectif est d'implanter des algorithmes parallèles de multiplication scalaire de point de courbe elliptique, inspirés de la démarche de Moreno et Hasan dans [48] :

— un *thread* producteur qui fournit les valeurs successives $2^i P$ par une séquence de doublements ;
— un *thread* consommateur qui accumule en les additionnant les valeurs produites par le *thread* producteur.

Il s'agit d'une configuration producteur-consommateur classique. Nous avons utilisé le même principe avec l'algorithme *Halve-and-add* (l'algorithme 2.9 page 71) pour les courbes elliptiques sur corps binaire. Nous appelons ces *threads* dans la suite respectivement *thread-producteur* et *thread-addition*.

— Dans une implantation parallèle, le *buffer* devient un emplacement en mémoire globale partagée. En conséquence, les *threads* devront communiquer via la mémoire cache de niveau L2 du processeur de la plate-forme envisagée, qu'on suppose partagée entre les différents cœurs. La taille de cette mémoire permet de stocker l'ensemble des valeurs de doublement et/ou de *halving* de points le cas échéant, même dans le cas des courbes sur corps de grande taille. Nous nous plaçons dans cette hypothèse.
— Le coût de la parallélisation n'est pas nul. Les dispositifs fournis par le système d'exploitation ne sont pas gratuits. Pour une bonne efficacité, nous devrons donc limiter le recours à ces dispositifs autant que possible.

7.2.1 Parallélisation

7.2.1.1 Synchronisation entre *threads*

Les deux cas que nous examinons (*Right-to-left Double-and-add*, l'algorithme 2.5 et *Halve-and-add*, l'algorithme 2.9) présentent donc des configurations producteur-consommateur classiques.

La manière la plus sûre de garantir un résultat de calcul correct est d'utiliser un dispositif de synchronisation rigoureux, effectuant les calculs par lots de taille réduite comme suggéré par Moreno et Hasan [48] :

— le *thread-producteur* calcule et stocke en mémoire partagée un lot de doublements ou divisions par deux de points ;
— lorsque le lot est complet, le *thread-producteur* envoie un signal au *thread* d'additions ;
— au signal reçu, le *thread* d'additions effectue le calcul avec les valeurs du lot produites et stockées en mémoire partagée ;
— le *thread-producteur* redémarre un nouveau lot en parallèle.

Le *thread-addition* doit attendre la fin de chaque lot pour effectuer les additions correspondantes et ceci est conforme au processus décrits par différentes auteurs : Tannenbaum dans [63] et Mueller dans [49].

Dans notre cas, nous sommes face à un dilemme :

— si la taille des lots est petite (ce que suggèrent Moreno et Hasan), nous calculerons le maximum d'additions en parallèle mais d'un autre côté, le coût de la synchronisation s'avère vite prohibitif ;

— si on augmente la taille des lots, le dernier lot traité par le *thread-addition* ne profitera d'aucun parallélisme, le *thread-producteur* sera mis en attente de façon improductive.

Nous devons aussi tenir compte de ce que la granularité des opérations d'addition, doublement ou *halving*, est petite en comparaison du coût de synchronisation. Ces opérations ne durent en effet que quelques centaines de cycles, soit quasiment le même ordre de grandeur que les barrières ou signaux. On rappelle ici les trois méthodes à notre disposition pour la synchronisation des *threads* (pour plus de détails, le lecteur se reportera à [13]) :

— `mutex`. Ce terme signifie exclusion mutuelle. C'est un verrou fourni par la bibliothèque `pthread` qui permet la synchronisation. Si un *thread* tient un `mutex`, un autre *thread* qui essaie de s'en saisir se vérouille et se met en attente jusqu'à la libération du `mutex` par le premier *thread*. L'utilisation des `mutex` est généralement réservée à la protection de sections critiques de codes. Le coût d'un verrouillage ou d'un déverrouillage est de l'ordre de 150 à 200 cycles d'horloge, ce qui est presque négligeable.

— `signals` : ils sont utilisés dans la communication inter-*thread* et inter-processus. Un *thread* en attente d'un signal est placé en sommeil par le système d'exploitation jusqu'à ce qu'un autre *thread* le réveille par l'envoi du signal attendu. Alors, le *thread* se réveille et continue son exécution. L'état de sommeil permet l'économie des ressources qui sont disponibles pour d'autres processus. D'après nos expériences et sur notre plate-forme, le coût de l'envoi d'un signal est de l'ordre de 2000 cycles d'horloge, mais peut varier selon les impératifs du système d'exploitation.

— `busy-waiting` : cette méthode consiste à utiliser une variable en mémoire globale et à l'utiliser comme drapeau pour maintenir le *thread-addition* dans une boucle en attendant que le *thread-producteur* en change la valeur. Le nom *busy-waiting* fait référence au fait que le *thread* est occupé à tourner dans la boucle dans une attente active. L'inconvénient principal de cette méthode est le gaspillage des ressources systèmes.

Nous avons expérimenté ces trois méthodes. Les `signals` s'avèrent trop coûteux comparés aux autres méthodes. Les techniques `busy-waiting` et `mutex` sont d'une efficacité assez similaire, mais l'utilisation de `mutex` est généralement meilleure. Nous fournissons table 7.1 des résultats d'implantations comparatives d'un algorithme parallèle *Right-to-left Double-and-add* qui illustre cette hiérarchie des performances respectives de ces trois approches sur notre plate-forme. Nous avons décidé en conséquence d'employer exclusivement la méthode basée sur l'emploi de `mutexes`.

Méthode de synchronisation	`mutex`	`busy-waiting`	`signals`
Right-to-left Double-and-add courbe NIST B233	154121	168195	184340

TABLE 7.1 – Performances comparées d'une multiplication scalaire parallèle en fonction des approches de synchronisation entre *threads*, en nombre de cycles d'horloge, processeur Intel Core i7, compilateur gcc 4.6.3.

7.2.1.2 Processus de synchronisation proposé.

Notre objectif est de trouver une façon d'éviter au maximum l'emploi de `mutex`. Le dispositif le plus léger, c'est de n'en utiliser qu'un seul : au tout début du calcul, le *thread-producteur* se saisit du `mutex`, ce qui place le *thread-addition* en attente immédiatement après sa création durant le calcul d'un premier lot de doublements (ou *halving* de points). À l'issue du calcul de ce premier lot, le *thread-producteur* relâche le `mutex` et continue la production des valeurs sans aucun autre verrouillage. Le *thread-addition* libéré commence les accumulations. Cette approche est montrée dans la figure 7.1.

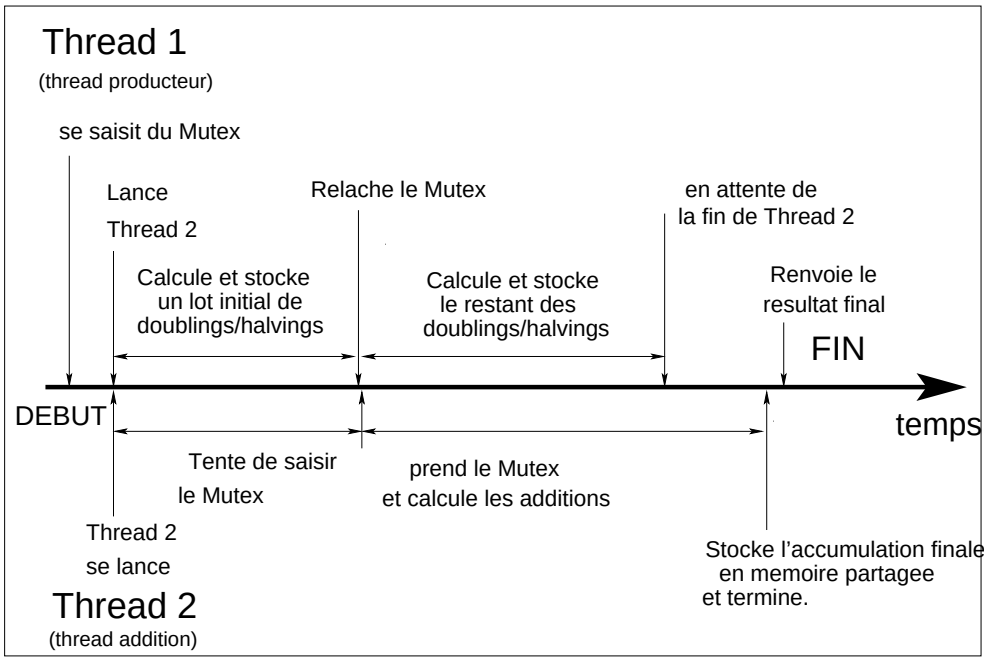


FIGURE 7.1 – Processus de synchronisation des *threads* dans notre implantation parallèle

Cependant, le résultat final n'est correct que si le dimensionnement du premier lot de points calculés avant la libération du `mutex` est suffisant, afin d'assurer l'antériorité de l'écriture en mémoire globale de toutes les valeurs de doublements/divisions par deux sur les calculs effectués par le *thread-addition*. Si le lot initial est trop petit, et à supposer que le poids de Hamming de la représentation du scalaire soit faible, la consommation des valeurs par le *thread-addition* risque d'être trop rapide, jusqu'à violer la dépendance lecture-après-écriture. Le résultat final de la multiplication scalaire est alors erroné. Pour éviter cette configuration, nous avons réglé la taille du lot de façon expérimentale, pour limiter le taux d'erreurs à une valeur aussi proche que possible de zéro. Dans nos résultats, ce taux est toujours inférieur à 1 %. Nous avons opté pour ce compromis dans le but de limiter le temps de calcul du lot initial par le *thread-producteur* et de profiter au maximum de la parallélisation. Cependant, les erreurs restantes se doivent d'être détectées et traitées, c'est obligatoire.

Pour l'élimination de ces erreurs résiduelles, nous avons ajouté un test dans le *thread-addition*. Le *thread-producteur* utilise à cet effet un compteur de boucle en mémoire partagée, accessible au *thread-addition*. Ceci permet de contrôler si le point utilisé dans une addition a bien été écrit en mémoire globale ou non par le *thread-producteur*. Si le coût de ce test n'est pas entièrement gratuit, il est pratiquement négligeable. Lorsque une erreur est détectée, c'est à dire lorsqu'une violation de la dépendance lecture-après-écriture est apparue, nous interrom-

pons le calcul du *thread-addition* et lançons un calcul de secours séquentiel de $k \cdot P$. En raison du faible taux d'erreurs, en pratique, les corrections n'ont pas d'effets significatifs sur les temps de calculs en moyenne.

L'algorithme 7.2 présente cette approche dans le cas *Right-to-left Double-and-add*, montrant en particulier le traitement des violation de dépendances.

Remarque 7.2.1. *Nous avions conclu la section 2.3.4.1 sur la moindre efficacité de l'algorithme* Right-to-left Double-and-add *par rapport à son homologue* Left-to-right Double-and-add *à cause de l'utilisation dans ce deuxième cas de l'addition en coordonnées projectives de points de la courbe elliptique au lieu de l'addition mixée. La complexité inférieure de la deuxième rend la version* Left-to-right Double-and-add *plus efficace dans le cas séquentiel. Le lecteur peut se reporter aux tables 2.7 page 61 et 2.12 page 67 pour les complexité respectives des différentes additions de points. Nous pouvons en déduire que ceci va néanmoins réduire le bénéfice de la parallélisation.*

En revanche, pour les courbes elliptiques sur corps binaires, le cas de l'algorithme Halve-and-add *parallèle est plus favorable. En effet, nous pouvons effectuer la séquence des* halvings *en coordonnées* $\lambda - representation$*, si nécessaire convertir le point en coordonnées affines pour un coût d'une multiplication, et utiliser ensuite une addition de point en coordonnées mixées.*

7.2.1.3 Choix de la représentation du scalaire

Dans les calculs séquentiels, la technique de recodage NAF permet une accélération sensible des calculs. Nous avons présenté cette technique dans la section 2.3.4.2. Dans les algorithmes parallèles, la situation est un peu différente. En effet, les réécritures NAF et γ-NAF diminuent le nombre d'additions qui, dans notre cas, sont effectuées par le *thread-addition* en parallèle. On le voit lorsque l'on analyse la quantité de calculs de chacun des *threads*. On peut évaluer cette quantité à l'aide des complexités fournies dans la table 2.13 page 73, dans la table 2.7 pour les courbes sur corps premiers $E(\mathbb{F}_p)$ et dans la table 2.12 dans le cas de courbes $E(\mathbb{F}_{2^m})$. Dans un souci de simplicité, nous avons supposé ici que l'élévation au carré $S = 0.8M$ dans $\mathbb{F}_p$ et nous avons négligé le carré et la racine carrée dans $\mathbb{F}_{2^m}$. Nous avons aussi supposé que le calcul de la *Half-Trace* pour l'opération de *halving* de point avait un coût équivalent à la multiplication. Les complexités qui en résultent apparaissent dans la table 7.2.

	Double-and-add sur $\mathbb{F}_p$			*Double-and-add* sur $\mathbb{F}_{2^m}$			*Halve-and-add* sur $\mathbb{F}_{2^m}$		
Repr.	*thread-* prod.	*thread-* add.	post-calculs	*thread-* prod.	*thread-* add.	post-calculs.	*thread-* prod.	*thread-* add.	post-calculs
binary	$6.2tM$	$5.1tM$	0	$4tM$	$6.5tM$	0	$2tM$	$4tM$	0
NAF	$6.2tM$	$3.4tM$	0	$4tM$	$4.33tM$	0	$2tM$	$2.66tM$	0
γ-*NAF* ($\gamma = 4$)	$6.2tM$	$2.04tM$	$33M$	$4tM$	$2.6tM$	$39M$	$2tM$	$1.6tM$	$39M$

TABLE 7.2 – Complexité des algorithmes parallèles à deux *threads* dans $E(\mathbb{F}_{2^m})$, pour un scalaire de taille t bits en représentation binaire, NAF et γ-NAF, en nombre de multiplications.

Dans la table 7.2, nous remarquons de façon générale que la complexité du *thread-addition* est supérieure à celle du *thread-producteur* pour une représentation binaire du scalaire. Lorsqu'on utilise la représentation NAF, les deux complexités s'équilibrent globalement. En revanche, dans le cas de la représentation γ-NAF, la quantité de calculs dans le *thread-addition* devient inférieure à celle du *thread-producteur*. Cela signifie qu'avec ce type de réécriture, le *thread-addition* progresse plus rapidement que le *thread-producteur* et devra donc même être mis en attente dans certains cas. Cependant, dans tous les cas, le *thread-addition* devra se terminer après le *thread-producteur*. De plus, pour l'approche γ-NAF, le retard dû aux calculs de

reconstruction (colonne Reconstruction), qui ne peut prendre place qu'après la fin de toutes les additions, vient encore pénaliser cette configuration.

Ces remarques sont confirmées par les chronogrammes donnés dans la figure 7.2 qui montrent les différents chronométrages de chaque *threads* dans les trois cas de représentation du scalaire. Ces chronogrammes correspondent à l'algorithme parallèle basé sur l'approche *Halve-and-add* pour une multiplication scalaire sur $E(\mathbb{F}_{2^{233}})$. Ces éléments nous conduisent à opter pour la réécriture NAF.

En complément de ce choix, et dans le but d'améliorer l'efficacité, nous avons mis au point une taille variable pour le lot initial de doublements/*halving*. En effet, le nombre d'additions effectuées par le *thread-addition* dépend du poids de Hamming de la représentation NAF du scalaire (le nombre de chiffres non nuls). Comme nous l'avons déjà mentionné, une violation de dépendance lecture-après-écriture peut apparaître si la taille du lot est trop petite. En revanche, si le poids de Hamming du scalaire est plus grand, le risque d'une telle violation diminue, et on pourrait alors réduire la taille du lot initial dans ce cas. Cette amélioration s'applique uniquement dans le cas où le *thread-addition* a un temps d'exécution similaire à celui du *thread-producteur* (cf. la table 7.2). Nous l'appliquons donc aux approches suivantes exclusivement :

— *Right-to-left Double-and-add* sur $\mathbb{F}_p$;
— *Right-to-left Double-and-add* sur $\mathbb{F}_{2^m}$.

À titre d'exemple, en notant $\mathcal{H}$ le poids de Hamming de la représentation NAF du scalaire, la taille du lot initiale dans le cas *Right-to-left Double-and-add* sur $\mathbb{F}_{2^{233}}$ (courbe NIST B233) est

$$\mathcal{IBS} = 16 - 1,5 \times (\mathcal{H} - 75).$$

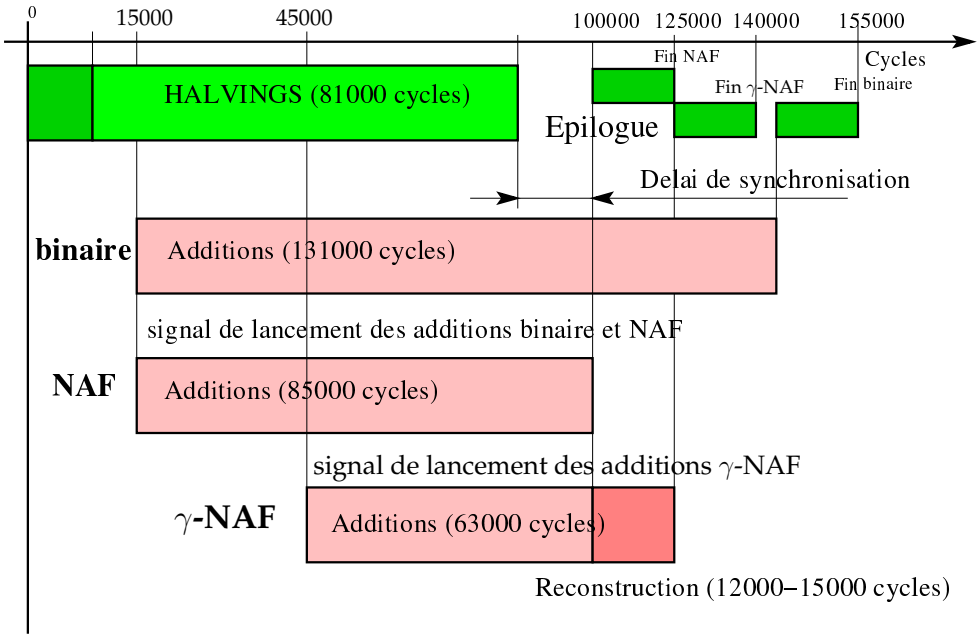


FIGURE 7.2 – Chronogramme des exécutions de l'approche *halve-and-add* parallèle avec un scalaire en représentation binaire, NAF et γ-NAF, sur courbe B233

Le résultat de ces réflexions conduit à l'élaboration de l'algorithme 7.2 basé sur l'approche *Right-to-left Double-and-add*, qui s'applique au cas des courbes sur corps premier et binaire, et de l'algorithme 7.3 basé sur l'approche *Halve-and-add*, pour les courbes sur corps binaires.

Dans ces algorithmes, on note $glbM.i$ le compteur de boucle du *thread-producteur* en mémoire globale.

Algorithme 7.2 *NAF Double-and-add* parallèle dans $E(\mathbb{F}_q)$

Require: scalaire $k, P \in E(\mathbb{F}_q)$.
Ensure: kP
1: Réécriture NAF de $k = (k_{t-1}, \ldots, k_1, k_0)$
2: Calcul de $\mathcal{IBS}$ en fonction de $\mathcal{H}$, poids de Hamming de NAF(k)
(Barrier)

// Calcul des Doublements (*thread-producteur*)

3: $D[0] \leftarrow P$
4: **for** $glbM.i = 1$ to $\mathcal{IBS}$ **do**
5: //*Doubling* LD projective
$D[glbM.i] \leftarrow D[glbM.i-1] \times 2$
6: **end for**
7: Signal vers *thread-addition*
8: **for** $glbM.i = \mathcal{IBS} + 1$ to $M - 1$ **do**
9: //*Doubling* LD projective
$D[glbM.i] \leftarrow D[glbM.i-1] \times 2$
10: **end for**

// Calcul des additions (*thread-addition*)

11: $Q \leftarrow \mathcal{O}$
12: Attente du signal du *thread-producteur*
13: **for** $i = 0$ to $M - 1$ **do**
14: **if** $i > glbM.i - 1$ **then**
15: Lancement calcul de secours ($Q \leftarrow kP$)
16: break
17: **end if**
18: **if** $k_i \neq 0$ **then**
19: //addition en coordonnées projectives
$Q \leftarrow Q + k_i D[i]$
20: **end if**
21: **end for**

(Barrier)
22: **return** Q

7.2.1.4 Version à quatre *threads* pour courbe sur $\mathbb{F}_{2^m}$

Nous rappelons l'approche parallèle de Taverne *et al.* dans [65] pour la multiplication scalaire de point de courbe elliptique sur corps binaire, que nous avons présentée dans la section 2.3.4.4. Elle se formule par l'équation suivante qui sépare le scalaire de taille t-bit en deux parties $k = k_1 + k_2$ comme suit :

$$k = \underbrace{(k'_t 2^{t-\ell} + \ldots + k'_\ell)}_{k_1} + \underbrace{(k'_{\ell-1} 2^{-1} + \ldots + k'_0 2^{-\ell})}_{k_2}. \tag{7.1}$$

En général, ℓ est proche de $t/2$ et représente la taille de k_2, soit la longueur de la multiplication scalaire effectuée par l'approche *Halve-and-add*. Le calcul s'effectue en parallèle par un *thread* calculant $k_1 P$ avec l'approche *Double-and-add* et un *thread* secondaire calculant $k_2 P$ comme rappelé plus haut.

Nous proposons de combiner l'approche de Taverne *et al.* et les approches parallèles présentées dans la section 7.2.1. Le résultat est un algorithme à quatre *threads*. La multiplication scalaire partielle $k_1 P$ est calculée avec l'approche de l'algorithme 7.2 à l'aide de deux *threads*, et $k_2 P$ est calculé avec l'algorithme 7.3, parallèle à deux *threads* également.

Par ce procédé, nous augmentons le niveau de parallélisme, mais cela requiert également des lancements de *threads* supplémentaires. Il est donc naturel que cette approche fonctionne mieux sur les courbes sur corps de la plus grande taille.

Notre implantation logicielle de cette approche à quatre *threads* (voir la figure 7.3) se base sur les stratégies suivantes pour la création des *threads* et la synchronisation :

Algorithme 7.3 *NAF Halve-and-add* parallèle dans $E(\mathbb{F}_{2^m})$

Require: scalaire k, $P \in E(\mathbb{F}_{2^m})$, $\mathcal{IBS}$.
Ensure: kP

1: Réécriture : $k' = k \cdot 2^{t-1} \bmod r$.
2: Réécriture NAF de $k' = (k'_{t-1}, \ldots, k'_1, k'_0)$
(Barrier)

Calcul des *halvings* (*thread-producteur*)

3: $H[0] \leftarrow P$

4: **for** $glbM.i = t-1$ to $t-1-\mathcal{IBS}$ **do**
5: //*Halving* en λ-représentation
$H[glbM.i] \leftarrow H[glbM.i-1] \times 2$
6: **end for**
7: Signal vers *thread-addition*.
8: **for** $glbM.i = t-2-\mathcal{IBS}$ to 0 **do**
9: //*Halving* en λ-représentation
$H[glbM.i] \leftarrow H[glbM.i-1] \times 2$
10: **end for**

Calcul des additions (*thread-addition*)

11: $Q \leftarrow \mathcal{O}$
12: Attente du signal du *thread-producteur*

13: **for** $i = 0$ to $M-1$ **do**
14: **if** $i > glbM.i - 1$ **then**
15: Lancement calcul de secours ($Q \leftarrow kP$)
16: break
17: **end if**
18: **if** $k_i \neq 0$ **then**
19: $H[i] \leftarrow$ conversion de $H[i]$ en affines
20: //addition en coordonnées mixées
$Q \leftarrow Q + k'_i H[i]$
21: **end if**
22: **end for**

(Barrier)
23: **return** Q

— Les *threads* sont lancés dans l'ordre suivant :
  1) le *thread* 1 (principal) est le *thread-producteur Halve-and-add*, qui lance le *thread* 2 ;
  2) le *thread* 2 correspond au *thread-producteur Double-and-add*, qui lance le *thread* 3 ;
  3) le *thread* 3 est le *thread-addition Halve-and-add* qui lance le *thread* 4 ;
  4) le *thread* 4 est le *thread-addition Double-and-add*.
— La réécriture du scalaire en deux parties est effectuée par le *thread* 3, *thread-addition Halve-and-add*, avant le lancement du *thread* 4.
— En raison du temps passé au lancement des *threads* et du calcul de la réécriture du scalaire, l'usage de `mutex` n'est pas nécessaire dans ce cas particulier. Il n'y a donc pas de lots initiaux dans les *threads* producteurs.
— Pour parer les violations de dépendances lecture-après-écriture dues aux défaillances de synchronisation, nous utilisons la même méthode que celle décrite dans la section 7.2.1.2. Ainsi, nous avons deux compteurs de boucles en mémoire partagée pour chacun des *threads* producteurs. Chaque *thread-addition* respectif contrôle l'absence de violation de dépendance, et le cas échéant, lance un calcul séquentiel partiel.

7.2.2 Performances

7.2.2.1 Stratégies d'implantation parallèle de la multiplication scalaire

Dans cette section, après une revue rapide des stratégies d'implantation des opérations sur les corps finis, nous présentons les performances des implantations logicielles des algorithmes parallèles que nous venons de voir, prenant en compte les contraintes pour une telle programmation concurrente.

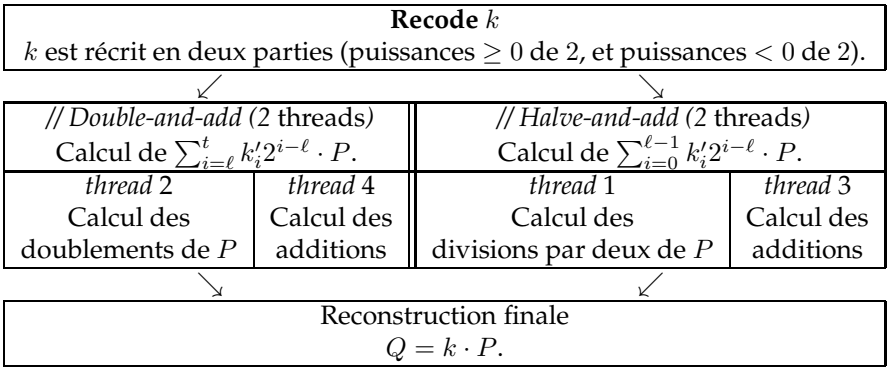


FIGURE 7.3 – Algorithme de multiplication scalaire dans $E(\mathbb{F}_{2^m})$ à 4 *threads*.

7.2.2.1.1 Stratégies d'implantation sur corps premier $\mathbb{F}_p$

Comme nous l'avons mentionné, le corps premier retenu est $\mathbb{F}_p$ avec $p = 2^{255} - 19$, qui a été introduit par Bernstein dans [7]. Pour le calcul des opérations sur le corps, nous avons utilisé le code librement disponible publié par Adam Langley dans [40]. Ce code s'avère plus efficace en comparaison de l'utilisation des fonctions de bas niveau de la bibliothèque GMP [2] pour le corps considéré. En effet, dans le code de Langley, les grands entiers sont stockés dans des mots de 64 bits contenant chacun 51 bits seulement. Ceci permet une meilleure prise en compte des retenues dans les additions et soustractions. Les multiplications et élévations au carré intègrent la réduction modulaire en effectuant directement l'opération

$$A \bmod p \equiv 19 \cdot A_H + A_L \bmod p,$$

avec $A = A_H \cdot 2^{255} + A_L$, comme nous l'avons présenté dans la section 1.1.3.4.2. En effet, les produits partiels de mots correspondants à A_H sont réduits à la volée, et les retenues éventuelles stockées dans les bits de garde de chaque mot avant réduction finale. Le carré utilise l'astuce que nous avons mentionnée en section 1.1.3.1 pour diminuer le nombre de produits élémentaires de mots. L'inversion modulaire est effectuée selon la méthode présentée dans la section 1.1.4.2, par une succession de carrés et de multiplications modulaires.

Nous avons réalisé des implantations sur deux courbes différentes, dont nous donnons les paramètres Annexes 8.1.2 et 8.1.3 :

— une courbe de Weierstrass (voir la section 2.3.1.1) ;
— une courbe Jacobi Quartic (voir la section 2.3.2.3).

7.2.2.1.2 Stratégies d'implantation sur corps binaire $\mathbb{F}_{2^m}$

Les courbes elliptiques sont celles du NIST dans [19], sur les corps $\mathbb{F}_{2^{233}} = \mathbb{F}_2[x]/(x^{233} + x^{74} + 1)$ et $\mathbb{F}_{2^{409}} = \mathbb{F}_2[x]/(x^{409} + x^{87} + 1)$. Nous avons déjà utilisé ces courbes dans les implantations présentées dans les chapitres précédents. Nous avons donc réutilisé strictement les mêmes codes qu'au chapitre 4, avec l'optimisation d'opérations combinées $AB + CD$ présentée Section 5.2.

Nous donnons les paramètres de ces deux courbes en annexe 8.1.1 pour les courbes sur corps binaires B233 et B409.

7.2.2.2 Expérimentations

Nous présentons maintenant les résultats expérimentaux de performances obtenus dans l'exécution de nos implantations logicielles.

La plate-forme utilisée est un ordinateur de bureau Optiplex 990 DELL® avec le système d'exploitation Linux, distribution Ubuntu 12.04. Le processeur est l'Intel core i7®-2600 Sandy Bridge 3.4GHz, comportant 4 cœurs physiques, ce qui correspond au nombre maximum de *threads* dans les algorithmes que nous expérimentons. Le code est écrit en langage C et compilé à l'aide de `gcc` version `4.6.3`.

En raison de la possibilité de préemption du système d'exploitation dans l'ordonnancement des processus, et pour éviter des difficultés dans l'affectation des cœurs aux différents *threads* que nous lançons, nous avons choisi d'exécuter nos codes dans une console de maintenance (*recovery mode*). Dans cette configuration, le système d'exploitation ne lance qu'un minimum de services. Cependant, dans des conditions normales de fonctionnement du système, nous avons noté un bon comportement général de nos codes. Toutefois, des perturbations aléatoires peuvent apparaître, et les exécutions sont moins régulières.

		Corps binaire $\mathbb{F}_{2^m}$		**Corps premier $\mathbb{F}_p$**	
		B233	**B409**	**Weierstrass**	**Jacobi Quartic**
Références					
Séquentiel 1 *thread*	*Double-and-add*	159000 (γ-NAF, $\gamma = 4$)	706000	256631 (NAF)	222558
	Halve-and-add	135000 (γ-NAF, $\gamma = 4$)	534000	-	-
2 *threads*	*Dbl/Hlv-and-add*	98000 (γ-NAF, $\gamma = 4$)	347000	-	-
Parallel NAF Double-and-add					
2 *threads*	moyenne	**154621**	**598491**	**218606**	**184048**
	Doublings	114713	505662	168958	134398
	Additions-D	120748	522869	125990	87415
Parallel NAF Halve-and-add					
2 *threads*	moyenne	**126639**	**430222**		
	Halvings	81630	300113		
	Additions-D	85107	373534		
Parallel NAF Double/halve-and-add					
4 *threads*	moyenne	**133273** ($\ell = 151$)	**324395** ($\ell = 246$)		
	Doublings	44672	202393		
	Additions-D	39333	200660		
	Halvings	67076	199625		
	Additions-H	55615	217534		

TABLE 7.3 – Performances des multiplications scalaires à 2 *threads* sur $E(\mathbb{F}_{2^m})$ et $E(\mathbb{F}_p)$, et 4 *threads* sur $E(\mathbb{F}_{2^m})$.

La table 7.3 présente les résultats de performance des implantations de la stratégie parallèle que nous proposons pour la multiplication scalaire de point de courbe elliptique.

Les mesures sont effectuées de la manière suivante : 100 échantillons de 2000 calculs, chaque échantillon pour un jeu de données différent aléatoire (point P et scalaire k) sont lancés. La meilleure performance de chaque échantillon est retenue et la valeur indiquée dans la table 7.3 est la moyenne de ces meilleures performances. Cette procédure permet de prendre en compte les variations de poids de Hamming des représentations des scalaires.

Pour chaque mesure, nous indiquons également la durée détaillée de chaque *thread*. On remarque qu'en général, le calcul se termine quelques milliers de cycles après la fin des calculs de doublements/*halvings* dans le *thread-producteur*. Ce décalage correspond au départ tardif du *thread-addition* (du fait du lot initial de points écrits en mémoire partagée par le *thread-*

producteur) et au temps de gestion de la synchronisation des *threads*. Dans le cas de l'approche à quatre *threads*, la valeur indiquée pour ℓ est la taille du scalaire k_2 pour le calcul parallèle *Halve-and-add* (cf. equation (7.1)). Nous avons estimé le surcoût dû à la gestion des violations de dépendance lecture-après-écriture : en moyenne, cela représente 2 à 6 % du temps total de calcul.

Les résultats pour la courbe B233 sur $\mathbb{F}_{2^{233}}$ font apparaître que la version 4 *threads* n'est pas compétitive. On peut l'attribuer au coût de création des *threads*. L'amélioration des approches deux *threads* pour cette courbe est également modeste.

La situation est meilleure sur le corps de plus grande taille $\mathbb{F}_{2^{409}}$. On note une accélération de 6,6% de notre approche en comparaison des résultats de l'approche de Taverne *et al.* γ*-NAF Double/halve-and-add*, soit 324395 cycles contre respectivement 347000 cycles. Les améliorations sont aussi plus significatives pour les approches à deux *threads* : de 15% (cas *Double-and-add*) à 19,5% (cas *Halve-and-add*).

Enfin, sur corps premier $\mathbb{F}_p$, nous remarquons en premier lieu que la multiplication scalaire sur courbe Jacobi Quartic est plus rapide par rapport au calcul sur courbe de Weierstrass. Ceci corrobore les complexités que nous avons dans la table 2.7. Nous notons également que les approches deux *threads* procurent une amélioration des performances de 15% à 17% comparées à celles des approches séquentielles NAF *Double-and-add*.

Comparaison : nous donnons dans la table 7.4 quelques résultats trouvés dans la littérature. Sur $\mathbb{F}_p$, les travaux de Longa concernent des implantations sur Intel Core 2, sur le corps $p = 2^{256} - 189$. Hamburg, pour sa part, propose des implantations sur Intel Core i7 Sandy Bridge, sur corps $p = 2^{252} - 2^{232} - 1$, mais avec un scalaire de taille plus réduite. Les autres travaux correspondent à des implantations sur le même corps et la même plate-forme que celle que nous avons utilisée. Nous pouvons voir que dans le cas $E(\mathbb{F}_{2^{233}})$, notre approche présente des performances en retrait par rapport aux meilleurs résultats publiés. Dans le cas $E(\mathbb{F}_{2^{409}})$, notre approche améliore de 9,4 % le résultat de Taverne *et al.* [65]. Pour finir, les performances publiées par Hamburg sont meilleures que notre méthode. On remarque cependant que le corps utilisé par ce dernier est de taille plus réduite, ainsi que le scalaire. Nous pouvons donc dire que notre approche améliore les meilleurs résultats connus pour un niveau de sécurité de 128 bits.

	multiplication scalaire	Courbe	Securité	processeur	Méthode	Cycles
$E(\mathbb{F}_p)$	Hamburg [21]	Montgomery	126	Intel core i7 SB	Montg. ladder	153000
	Langley [40]	Curve25519	128	Intel core i7 SB	Montg. ladder	229000 [1]
	Bernstein [8, 7]	Curve25519	128	Intel core i7 SB	Montg. ladder	194000
	Longa *et al.* [42]	jac256189	128	Intel core 2 Duo	γ-NAF D&A	337000
	Longa *et al.* [42]	ted256189	128	Intel core 2 Duo	γ-NAF D&A	281000
	Ce travail	jac25519	128	Intel core i7 SB	//NAF D&A	184048
$E(\mathbb{F}_{2^m})$	Nègre *et al.* [51]	B233	112	Intel core i7 SB	γ-NAF D-H&A	98000
	Taverne *et al.* [65]	B233	112	Intel core i7 SB	γ-NAF D-H&A	102000
	Ce travail	B233	112	Intel core i7 SB	//NAF H&A 2 th.	126639
	Nègre *et al.* [51]	B409	192	Intel core i7 SB	γ-NAF D-H&A	347000
	Taverne *et al.* [65]	B409	192	Intel core i7 SB	γ-NAF D-H&A	358000
	Ce travail	B409	192	Intel core i7 SB	//NAF D-H&A 4 th.	324395

[1] compilé et exécuté sur notre plate-forme.

TABLE 7.4 – Comparaison des performances avec l'état de l'art.

7.3 Conclusion de ce chapitre

Dans ce travail, nous avons réalisé des implantations logicielles d'algorithmes parallèles de multiplication scalaire kP sur $E(\mathbb{F}_{2^m})$ et $E(\mathbb{F}_p)$. Nous avons considéré la parallélisation suggérée par Moreno et Hasan dans [48], qui utilise l'approche *Right-to-left Double-and-add* et sépare le calcul en deux *threads* : un *thread-producteur* calculant et écrivant en mémoire globale les valeurs 2^iP ou $2^{-i}P$ pour $i = 1, \ldots, t$ et un *thread-addition* qui accumule ces valeurs pour obtenir kP. Nous avons proposé un dispositif le plus léger possible pour la synchronisation entre les deux *threads*. En complément, pour éviter les erreurs de calcul dues aux violations de dépendance, nous avons proposé une méthode d'auto-contrôle de ces violations avec calcul de secours séquentiel. Dans le cas particulier des courbes sur corps binaire $E(\mathbb{F}_{2^m})$, nous avons combiné cette approche avec celle de Taverne *et al.* [65] (*Double/halve-and-add*) pour construire un algorithme à 4 *threads*.

Les résultats expérimentaux montrent que ces techniques de parallélisation procurent une accélération sur les calculs de multiplication scalaire en comparaison des meilleurs résultats précédents publiés dans la littérature. Dans la plupart des cas, l'amélioration est d'environ 15% sur le temps de calcul.

Nous remarquons que l'approche parallèle que nous avons implantée reprend les principales caractéristiques de la proposition de Moreno et Hasan en matière de résistance à l'attaque SPA :

— régularité des opérations du *thread-producteur*, nous stockons en effet tous les points en mémoire partagée ;
— régularité des opérations d'additions, sauf en cas de violation de dépendance ;
— limitation identique due à la possibilité de connaître le poids de Hamming du scalaire ;

Notre travail a montré la possibilité d'implanter l'algorithme parallèle de Moreno et Hasan de façon compétitive. Les performances sont au niveau de l'état de l'art, mais c'est au prix d'un dispositif de synchronisation qui n'empêche pas les violations de dépendances. La régularité de nos implantations peut donc être entachée pour certains scalaires. La possibilité de tirer parti de l'information donnée par la détection de ces violations par une attaque ciblée est à évaluer. À cette réserve près, notre implantation conserve les caractéristiques de l'approche de Moreno et Hasan en terme de résistance à l'attaque SPA.

Nous terminons par deux questions, qui pourront faire l'objet de futurs travaux. En premier lieu, la question de la résistance à la *Timing Attack* reste ouverte, et il faudrait examiner en détail les opérations d'addition, de doublement et de *halving* de points ainsi que les opérations sur les corps finis pour le vérifier. En second lieu, en matière de résistance à l'attaque DPA, Moreno et Hasan proposent, en extension de leur approche, de randomiser l'ordre dans lequel on effectue les additions. Cette technique pourrait également s'appliquer à notre cas, mais la pénalité en performance pour assurer le respect des dépendances lecture-après-écriture reste à évaluer.

Conclusion

La cryptographie asymétrique s'est répandue depuis la fin des années 1970 à la suite des travaux de Diffie et Hellman [17] et de la diffusion du protocole RSA dû à Rivest, Shamir et Adleman [54]. La sécurité de ces protocoles repose sur la difficulté de problèmes mathématiques sous-jacents tels que le problème de la factorisation des grands entiers ou le problème du logarithme discret. Aujourd'hui, la difficulté de ces problèmes est considérée comme étant compatible avec les exigences de sécurité. Ces protocoles permettent d'échanger des clés secrètes, de crypter des messages et de les signer (authenticité, intégrité et non répudiation). Ils mettent en œuvre des opérations arithmétiques sur corps finis telles que l'exponentiation modulaire ou la multiplication scalaire de point de courbe elliptique sur corps fini. Si les protocoles sont sûrs sur le plan calculatoire en raison de la difficulté des problèmes mathématiques que nous venons d'évoquer, la possibilité d'exploiter des informations recueillies sur un canal auxiliaire permet de contourner la difficulté de ces problèmes. Ainsi, un attaquant peut retrouver les données secrètes (clé de déchiffrement en particulier). Par exemple, à l'aide d'une mesure unique du courant électrique consommé par un appareil effectuant des calculs cryptographiques avec un algorithme non protégé, l'attaque *Simple Power Analysis* permet de retrouver l'exposant d'une exponentiation modulaire.

Au cours de ce travail de thèse, nous avons étudié l'élaboration de différents algorithmes d'opérations arithmétiques effectuées par les protocoles de cryptographie asymétrique, tout en apportant ou conservant des propriétés de résistance à certaines attaques par canal auxiliaire, en particulier l'attaque *Simple Power Analysis*. Nous avons procédé à leur implantation logicielle efficace et avons vérifié leurs éventuels bénéfices en terme de performance.

Dans une première partie, nous avons présenté l'état de l'art en matière d'arithmétique des corps finis, d'opérations arithmétiques en usage dans la mise en œuvre des protocoles ainsi que les principales attaques par canal auxiliaire susceptibles de menacer les opérations précédentes.

Au chapitre 1, nous avons rappelé les fondements de l'arithmétique modulaire sur les anneaux et corps finis, en grande caractéristique et en caractéristique deux. Nous avons présenté les principales opérations sur les éléments de ces ensembles : l'addition, la soustraction, la multiplication et l'élévation au carré, la réduction modulaire, l'inversion modulaire et le calcul de racine carrée dans le cas de la caractéristique deux. Pour chacune de ces opérations, nous présentons les algorithmes les plus performants à notre connaissance.

Au chapitre 2, après avoir présenté les principaux protocoles, d'échange de clés, de chiffrement et de signature, basés sur le problème de la factorisation ou du logarithme discret, nous avons présenté les algorithmes d'exponentiation modulaire sur anneau ou corps fini $\mathbb{Z}/N\mathbb{Z}$: les deux approches *Right-to-left Square-and-multiply* et *Left-to-right Square-and-multiply*. Nous avons ensuite abordé les courbes elliptiques sur corps fini de grande caractéristique et de caractéristique deux. Nous avons rappelé la structure de groupe formée par l'ensemble des points muni d'une loi d'addition donnée par la méthode corde-tangente. Nous avons présenté quelques systèmes de coordonnées projectives qui permettent d'améliorer l'efficacité des calculs d'ad-

dition et de doublement. Dans le cas particulier de la caractéristique deux, nous avons aussi présenté l'opération de *halving*, soit la division par deux d'un point d'ordre impair de la courbe. Nous avons terminé par la présentation des approches les plus connues en matière de multiplication scalaire de point de courbe elliptique : *Double-and-add*, *Halve-and-add* et *Double/halve-and-add* (approche parallèle de Taverne *et al.* dans [64, 65]).

Au chapitre 3, nous avons abordé la question des attaques par canal auxiliaire. En préambule de ce chapitre, nous avons présenté une classification générale de ces attaques, leurs différentes applications et les menaces qui en découlent. Nous avons présenté l'attaque *Simple Power Analysis*, qui consiste à exploiter une mesure de consommation de puissance instantanée durant un calcul et qui révèle le scalaire secret par le repérage des motifs correspondant aux différentes opérations effectuées par la plate-forme. Nous avons présenté ensuite les principales contre-mesures de la littérature, les algorithmes réguliers suivants : l'algorithme *Double-and-add-always* et sa variante pour l'exponentiation modulaire, l'algorithme de multiplication scalaire régulière *Double-add* de Joye dans [30], l'algorithme d'exponentiation modulaire *Square-always* de Clavier *et al.* dans [15], l'échelle binaire de Montgomery et les approches *Regular left-to-right* 2^γ*-ary exponentiation* et *Regular right-to-left* 2^γ*-ary exponentiation* proposées par Joye et Tunstall dans [31]. Nous avons présenté ensuite deux attaques : la *Timing Attack* et la *Differential Power Analysis* et leurs principales contre-mesures.

Dans une deuxième partie, nous avons exposé nos différentes contributions. Les deux premiers chapitres de cette deuxième partie sont consacrés à des améliorations au niveau des opérations combinées sur les anneaux et corps finis, puis appliquées aux algorithmes d'exponentiation modulaire et/ou multiplication scalaire de point de courbe elliptique dans le but d'améliorer les performances. Le chapitre suivant propose une multiplication scalaire parallèle de point de courbe elliptique sur corps binaire basée sur l'échelle binaire de Montgomery. Le dernier chapitre de cette partie est consacré à l'implantation d'une approche parallèle basée sur les algorithmes *Double-and-add*, *Halve-and-add* et *Double/halve-and-add*.

Au chapitre 4, nous avons présenté des opérations combinées de type $A \cdot B, A \cdot C$ et $AB_1, \ldots, AB_\ell$, soient des multiplications modulaires de Montgomery partageant un opérande commun, dans un anneau fini $\mathbb{Z}/N\mathbb{Z}$. L'amélioration de la complexité est de l'ordre de 25 % pour l'opération combinée $A \cdot B, A \cdot C$ et jusqu'à 50 % sur les multiplications multiples $AB_1, \ldots, AB_\ell$. Nous avons appliqué ces opérations aux algorithmes réguliers d'exponentiation modulaire résistants face à l'attaque *Simple Power Analysis* suivants : l'échelle binaire de Montgomery et les deux algorithmes *Regular left-to-right* 2^γ*-ary exponentiation* et *Regular right-to-left* 2^γ*-ary exponentiation* proposés par Joye et Tunstall dans [31]. L'application de ces opérations combinées à ces algorithmes se traduit par des gains de complexité de 4% pour l'approche *Regular right-to-left* 2^γ*-ary exponentiation*, 8% pour son homologue *Regular left-to-right* 2^γ*-ary exponentiation* et de 13% pour l'échelle binaire de Montgomery, dans le cas d'une exponentiation RSA de taille 2048 bits. Ces algorithmes améliorés ont fait l'objet d'implantations logicielles qui apportent des gains en performance de 4 % dans le cas *Regular right-to-left* 2^γ*-ary exponentiation*, jusqu'à plus de 8 % dans le cas *Regular left-to-right* 2^γ*-ary exponentiation*, et jusqu'à 15 % dans le cas de l'échelle binaire de Montgomery, pour des exponentiations de taille 4096 bits.

Au chapitre 5, nous avons étudié l'impact d'opérations combinées de type $A \cdot B, A \cdot C$ et $A \cdot B + C \cdot D$ sur corps binaire $\mathbb{F}_{2^m}$ aux additions et doublements de points de courbe elliptique sur corps binaire et avons appliqué ensuite ces opérations de points sur les algorithmes de multiplication scalaire suivants : *Double-and-add*, *Halve-and-add* et *Double/halve-and-add* (approche parallèle de Taverne *et al.* dans [64, 65]) utilisant la réécriture du scalaire γ*-NAF*, ainsi que l'échelle binaire de Montgomery. Nous avons présenté la complexité de ces opérations combinées dans les cas de deux approches différentes pour la multiplication de polynômes en caractéristique deux, l'approche classique que nous avons désignée *CombMul* utilisant le jeu

d'instruction conventionnel (OU exclusif, ET et décalages), et l'approche de Karatsuba récursive, avec l'emploi d'une multiplication élémentaire de polynômes du jeu d'instruction AES des processeurs Intel Core iX. Nous avons combiné ces opérations améliorées avec des perfectionnements appelés *Lazy-reduction* et avons réalisé les implantations logicielles correspondantes sur deux plates-formes à base d'Intel Core 2 Duo et Intel Core i5 pour deux courbes elliptiques préconisées par le *National Institute for Standards and Technology* (B233 et B409). Nous avons alors obtenu des gains de performances significatifs sur toutes les plates-formes et toutes les courbes pour l'opération combinée $A \cdot B + C \cdot D$, de l'ordre de 4% pour l'approche γ-*NAF Double-and-add* et l'échelle binaire de Montgomery.

Au chapitre 6, nous avons présenté une approche parallèle de l'échelle binaire de Montgomery appliquée à la multiplication de point de courbe elliptique sur corps binaire. Nous avons proposé un nouvel algorithme d'échelle binaire de Montgomey basé sur l'opération de *halving*, présentant les propriétés de régularité conférant une résistance face à l'attaque *Simple Power Analysis*, et dans lequel aucune opération n'étant inutile, la propriété de détection d'erreur et de résistance à la *safe-error attack* reste préservée. Cette approche n'est pas efficace en soi en raison de la contrainte d'utilisation de coordonnées affines dans les additions de points, mais permet l'élaboration d'une approche parallèle de l'échelle binaire de Montgomery dans l'esprit de celle de Taverne *et al.* dans [64, 65] utilisant l'échelle binaire de Montgomery classique basée sur le doublement de point ainsi que notre approche à base de *halving*. Nous avons réalisé les implantations logicielles correspondantes de multiplication scalaire de point de courbe elliptique sur les deux courbes B233 et B409 sur deux plates-formes différentes : un processeur Intel Core i7 et une plate-forme mobile utilisant un processeur Qualcomm Snapdragon à deux cœurs basés sur l'architecture ARM7. Les gains de performances constatés de l'approche parallèle en comparaison de l'échelle binaire de Montgomery séquentielle basée sur le doublement de point sont : de 5 et 10,5 % sur la plate-forme Intel respectivement dans le cas B233 et B409, et de 19 % sur la plate-forme Snapdragon pour les deux courbes.

A chapitre 7, notre étude a porté sur l'implantation logicielle de l'approche parallèle de l'algorithme *Double-and-add* présentée par Moreno et Hasan dans [48]. Leur approche théorique vise à apporter de la résistance à l'attaque *Simple Power Analysis*. Notre implantation quant à elle a pour objectif d'améliorer les performances tout en préservant les qualités de résistance de leur algorithme. Nous avons pour ce faire travaillé sur deux axes : le choix du processus de synchronisation le plus adapté ainsi que son implantation, ce qui nous a amené à choisir une approche basée sur l'usage d'un unique dispositif d'exclusion mutuelle (*Mutex*), avec un contrôle en continu du respect des dépendances lecture-après-écriture pour garantir l'obtention d'un résultat correct, avec lancement d'un calcul séquentiel de secours en cas de violation ; le choix de la meilleure représentation du scalaire dans le contexte de cette parallélisation, à savoir la représentation NAF, et qui en donne la meilleure exploitation. Nous avons également combiné ces approches avec celle de Taverne *et al.* dans [64, 65] pour proposer un algorithme *Double/halve-and-add* à quatre threads. Nous avons réalisé les implantations logicielles de multiplications scalaires de point de courbe elliptique correspondantes : *Parallel NAF Double-and-add* sur courbes sur corps binaires B233 et B409, sur une courbe elliptique de Weierstrass et une courbe Jacobi Quartic sur corps $\mathbb{F}_p$ avec $p = 2^{255} - 19$, *Parallel NAF Halve-and-add* et *Parallel NAF Double-halve-and-add* à quatre threads pour les courbes B233 et B409. Les gains de performances sont modestes pour les différentes versions sur la courbe B233. En revanche, pour la courbe B409, les gains vont de 6,6 % pour la version *Parallel NAF Double-halve-and-add* à quatre threads en comparaison de l'approche à deux threads de Taverne *et al.*, et respectivement 15 et 19 % pour les versions *Parallel NAF Double-and-add* et *Parallel NAF Halve-and-add* par rapport aux versions séquentielles correspondantes. Enfin, pour les courbes sur corps premier, l'approche *Parallel NAF Double-and-add* présente une amélioration de 15 % pour la courbe de

Weierstrass et 17 % pour la courbe Jacobi Quartic, toujours en comparaison avec la version séquentielle correspondante.

Chapitre 8

Annexe

8.1 Annexe : Paramètres des courbes elliptiques

8.1.1 Courbes elliptiques sur corps binaires $\mathbb{F}_{2^m}$

Ces deux courbes sont tirées des préconisations du NIST dans [19].
L'équation de la courbe est :

$$y^2 + xy = x^3 + x^2 + b \text{ where } b \in \mathbb{F}_{2^m}.$$

Pour la courbe B233 :

$$\begin{aligned} a &= 1, \\ h &= 2, \\ f(x) &= x^{233} + x^{74} + 1, \\ b &= \texttt{0x00000066 647ede6c 332c7f8c 0923bb58 213b333b 20e9ce42 81fe115f 7d8f90ad}, \\ r &= \texttt{0x00000100 00000000 00000000 00000000 0013e974 e72f8a69 22031d26 03cfe0d7}. \end{aligned}$$

où le nombre de points de la courbe est $r \times h$.
Pour la courbe B409, nous avons :

$$\begin{aligned} a &= 1, \\ h &= 2, \\ f(x) &= x^{409} + x^{87} + 1, \\ b &= \texttt{0x0021a5c2 c8ee9feb 5c4b9a75 3b7b476b 7fd6422e f1f3dd67 4761fa99 d6ac27c8} \\ &\quad \texttt{a9a197b2 72822f6c d57a55aa 4f50ae31 7b13545f}, \\ r &= \texttt{0x01000000 00000000 00000000 00000000 00000000 00000000 000001e2 aad6a612} \\ &\quad \texttt{f33307be 5fa47c3c 9e052f83 8164cd37 d9a21173}. \end{aligned}$$

8.1.2 Courbe de Weierstrass sur corps premier $\mathbb{F}_p$

L'équation de la courbe est :

$$y^2 = x^3 - 3x + b \text{ où } b \in \mathbb{F}_p.$$

Les paramètres de la courbe sont :

$$\begin{aligned} p &= 2^{255} - 19 \\ b &= \texttt{0x1d09bac9ffe9e7f8284aed0442552779bcdef2e62b9cb1d568513fa798b94003} \\ r &= \texttt{0x800000000000000000000000000000012c18945a05ad7f2edf026258ea5288ef} \end{aligned}$$

r est l'ordre premier d'un point P.

8.1.3 Courbe Jacobi Quartic sur corps premier $\mathbb{F}_p$

L'équation de la courbe est :

$$y^2 = x^4 - \frac{3}{2}\theta x^2 + 1, \theta \in \mathbb{F}_p.$$

Les paramètres de la courbe sont :

θ = 0x1731beeea2156180446f9e5ab64af78d4ed3e0eb68d5070c10ef2468b910d5f7
nombre de points :
$h \times r$ = 0x8000000000000000000000000000002672bdbb41f31390c5527cab6e282744
= $4 \cdot$ 0x2000000000000000000000000000000099caf6ed07cc4e431549f2adb8a09d1

La courbe Jacobi Quartic est isomorphe à la courbe de Weierstrass suivante :

$$y^2 = x^3 + ax + b$$

où : $a = (-16 - 3\theta^2)/4$ et $b = -\theta^3 - a\theta$. De là, on tire :

a = 0xc500be2450246d16c114830a5d1aef9c2b80c567b4fd87562c69db659713ad2,
b = 0xa38f53e5d27462dcdada9a78b9eac482ef06e855af92ca704060c551a9a5854.

Bibliographie

[1] Performance Application Programming Interface (PAPI). http://icl.cs.utk.edu/papi/.

[2] The GNU Multiple Precision Arithmetic Library (GMP). http://gmplib.org/.

[3] Explicit Formula Database, 2014. http://www.hyperelliptic.org/EFD/.

[4] Diego F. Aranha, Julio López, and Darrel Hankerson. Efficient Software Implementation of Binary Field Arithmetic Using Vector Instruction Sets. In *LATINCRYPT*, volume 6212 of *LNCS*, pages 144–161. Springer, 2010.

[5] Roberto M. Avanzi, Nicolas Thériault, and Zheng Wang. Rethinking Low Genus Hyperelliptic Jacobian Arithmetic over Binary Fields : Interplay of Field Arithmetic and Explicit Formulæ. *J. Mathematical Cryptology*, 2(3) :227–255, 2008.

[6] Roberto Maria Avanzi and Nicolas Thériault. Effects of Optimizations for Software Implementations of Small Binary Field Arithmetic. In *WAIFI*, volume 4547 of *LNCS*, pages 69–84. Springer, 2007.

[7] Daniel J. Bernstein. Curve25519 : New diffie-hellman speed records. In *Public Key Cryptography - PKC, 9th International Conference on Theory and Practice of Public-Key Cryptography, Proceedings*, pages 207–228, 2006.

[8] Daniel J. Bernstein and Lange T. (eds). eBACS : ECRYPT Benchmarking of Cryptograhic Systems. http://bench.cr.yp.to/, 2012. accédé le 25 mai 2014.

[9] Jean-Louis Beuchat, Emmanuel López-Trejo, Luis Martínez-Ramos, Shigeo Mitsunari, and Francisco Rodríguez-Henríquez. Multi-core Implementation of the Tate Pairing over Supersingular Elliptic Curves. In *CANS*, volume 5888 of *LNCS*, pages 413–432, 2009.

[10] Olivier Billet and Marc Joye. The Jacobi Model of an Elliptic Curve and Side-Channel Analysis. In *AAECC*, pages 34–42, 2003.

[11] Antoon Bosselaers, René Govaerts, and Joos Vandewalle. Comparison of Three Modular Reduction Functions. In *Advances in Cryptology - CRYPTO, 13th Annual International Cryptology Conference, Santa Barbara, California, USA, August 22-26, Proceedings*, pages 175–186, 1993.

[12] Michael Brown, Darrel Hankerson, Julio López, and Alfred Menezes. Software Implementation of the NIST Elliptic Curves Over Prime Fields. In *Topics in Cryptology - CT-RSA, The Cryptographer's Track at RSA Conference, San Francisco, CA, USA, April 8-12, Proceedings*, pages 250–265, 2001.

[13] Christophe Blaess. *Programmation système en C sous Linux*. Eyrolles, 2009.

[14] Mathieu Ciet and Marc Joye. (Virtually) Free Randomization Techniques for Elliptic Curve Cryptography. In *Information and Communications Security (ICICS)*, volume 2836 of *LNCS*. Springer-Verlag, 2003.

[15] Christophe Clavier, Benoit Feix, Georges Gagnerot, Mylène Roussellet, and Vincent Verneuil. Square Always Exponentiation. In *Progress in Cryptology - INDOCRYPT - 12th*

International Conference on Cryptology in India, Chennai, India, December 11-14. Proceedings, pages 40–57, 2011.

[16] Jean-Sébastien Coron. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In *CHES*, pages 292–302, 1999.

[17] Whitfield Diffie and Martin Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6) :644–654, 1976.

[18] Kenny Fong, Darrel Hankerson, Julio López, and Alfred Menezes. Field Inversion and Point Halving Revisited. *IEEE Trans. Computers*, 53(8) :1047–1059, 2004.

[19] Patrick Gallagher and Cita Furlani. Digital Signature Standard (DSS). In *Federal Information Processing Standards Publications*, volume FIPS 186-3 of *Federal Information Processing Standards Publications (NIST)*, page 93. National Institute of Standards and Technology, 2009.

[20] Jorge Guajardo, Sandeep S. Kumar, Christof Paar, and Jan Pelzl. Efficient Software-Implementation of Finite Fields with Applications to Cryptography. *Acta Applicandae Mathematica*, 93(1-3) :3–32, 2006.

[21] Mike Hamburg. Fast and Compact Elliptic-Curve Cryptography. Technical report, Cryptology ePrint Archive, Report 309, 2012. http://eprint.iacr.org/.

[22] Darrel Hankerson, Julio López Hernandez, and Alfred Menezes. Software Implementation of Elliptic Curve Cryptography over Binary Fields. In *Cryptographic Hardware and Embedded Systems - CHES*, volume 1965 of *LNCS*, pages 1–24. Springer, 2000.

[23] Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2004.

[24] Mustapha Hedabou, Pierre Pinel, and Lucien Bénéteau. A Comb Method to Render ECC Resistant against Side Channel Attacks. *IACR Cryptology ePrint Archive*, page 342, 2004.

[25] Hüseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. Twisted Edwards Curves Revisited. In *ASIACRYPT*, pages 326–343, 2008.

[26] Hüseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. Jacobi Quartic Curves Revisited. Cryptology ePrint Archive, Report 312, 2009. http://eprint.iacr.org/.

[27] Toshiya Itoh and Shigeo Tsujii. A Fast Algorithm for Computing Multiplicative Inverses in $\mathbb{F}_{2^m}$ Using Normal Bases. *Information and Computation*, 78 :171–177, 1988.

[28] Joseph H. Silverman Jeffrey Hoffstein, Jill Pipher. *An Introduction to Mathematical Cryptography*. Springer Science+Business Media, LLC. Springer, 2008.

[29] John M. Pollard. A Monte Carlo Method for Factorization. *BIT Numerical Mathematics*, 15(3) :331–334, 1975.

[30] Marc Joye. Highly Regular Right-to-Left Algorithms for Scalar Multiplication. In *Cryptographic Hardware and Embedded Systems - CHES, 9th International Workshop, September 10-13, Proceedings*, pages 135–147, 2007.

[31] Marc Joye and Michael Tunstall. Exponent Recoding and Regular Exponentiation Algorithms. In *Progress in Cryptology - AFRICACRYPT*, volume 5580 of *LNCS*, pages 334–349. Springer, 2009.

[32] Marc Joye and Sung-Ming Yen. Checking before Output May Not Be Enough against Fault-based Cryptanalysis. *IEEE Trans. on Computer*, 49(9) :967–970, 2000.

[33] Kwang Ho Kim and So In Kim. A New Method for Speeding Up Arithmetic on Elliptic Curves over Binary Fields. Technical report, National Academy of Science, Pyongyang, D.P.R. of Korea, 2007.

[34] Judson Knight. Engulf, Operation, 2004. http://www.encyclopedia.com/utility/printdocument.aspx?id=1G2:3403300273.

[35] Erik Woodward Knudsen. Elliptic Scalar Multiplication Using Point Halving. In *ASIACRYPT*, pages 135–149, 1999.

[36] Neal Koblitz. Elliptic Curve Cryptosystems. *Mathematics of Computation*, 48 :203–209, 1987.

[37] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology - CRYPTO, 16th Annual International Cryptology Conference, August 18-22, Proceedings*, pages 104–113, 1996.

[38] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Advances in Cryptology, CRYPTO*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.

[39] Paul C. Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to Differential Power Analysis. *J. Cryptographic Engineering*, 1(1) :5–27, 2011.

[40] Adam Langley. C25519 Code, 2008. http://code.google.com/p/curve25519-donna/.

[41] Pierre-Yvan Liardet and Nigel P. Smart. Preventing SPA/DPA in ECC Systems Using the Jacobi Form. In *CHES*, number Generators in LNCS, pages 391–401, 2001.

[42] Patrick Longa and Catherine H. Gebotys. Efficient Techniques for High-Speed Elliptic Curve Cryptography. In *CHES*, pages 80–94, 2010.

[43] Julio López and Ricardo Dahab. Fast Multiplication on Elliptic Curves over $\mathbb{F}_{2^m}$ without Precomputation. In *CHES*, pages 316–327, 1999.

[44] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[45] Victor Miller. Use of Elliptic Curves in Cryptography. In *Advances in Cryptology, Proceedings of CRYPTO'85*, volume 218 of *LNCS*, pages 417–426. Springer, 1986.

[46] Peter Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation*, 44(170) :519–521, 1985.

[47] Peter Montgomery. Speeding the Pollard and Elliptic Curve Methods of Factorization. *Mathematics of Computation*, 48(177) :243–264, 1987.

[48] Carlos Moreno and M. Anwar Hasan. SPA-Resistant Binary Exponentiation with Optimal Execution Time. *J. Cryptographic Engineering*, 1(2) :87–99, 2011.

[49] Frank Mueller. A Library Implementation of POSIX Threads under UNIX. In *USENIX Winter*, pages 29–42, 1993.

[50] Christophe Nègre, Thomas Plantard, and Jean-Marc Robert. Efficient Modular Exponentiation Based on Multiple Multiplications by a Common Operand. In *22nd IEEE Symposium on Computer Arithmetic, ARITH, June 22-24*, pages 144–151, 2015.

[51] Christophe Nègre and Jean-Marc Robert. Impact of Optimized Field Operations *AB*, *AC* and *AB* + *CD* in Scalar Multiplication over Binary Elliptic Curve. In *Progress in Cryptology - AFRICACRYPT, 6th International Conference on Cryptology in Africa, June 22-24.*, LNCS, pages 279–296, 2013.

[52] Christophe Nègre and Jean-Marc Robert. New Parallel Approaches for Scalar Multiplication in Elliptic Curve over Fields of Small Characteristic. *IEEE Trans. Computers*, 64(10) :2785–2890, 2015.

[53] Christof Paar. A New Architecture for a Parallel Finite Field Multiplier with Low Complexity Based on Composite Fields. *IEEE Trans. on Comp.*, 45 :856, 1996.

[54] Ron Rivest, Adi Shamir, and Leonard Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *J. ACM*, 21(2) :120–126, 1978.

[55] Jean-Marc Robert. Parallelized Software Implementation of Elliptic Curve Scalar Multiplication. In *Information Security and Cryptology - 10th International Conference, Inscrypt, December 13-15, Revised Selected Papers*, LNCS, pages 445–462, 2014.

[56] Francisco Rodríguez-Henríquez, Guillermo Morales-Luna, Nazar A. Saqib, and Nareli Cruz Cortés. Parallel Itoh-Tsujii Multiplicative Inversion Algorithm for a Special Class of Trinomials. *IACR Cryptology ePrint Archive*, page 35, 2006.

[57] Richard Schroeppel. Elliptic Curve Point Halving Wins Big. In *Second Midwest Arithmetical Geometry in Cryptography Workshop*, Nov. 2000.

[58] Michael Scott. Optimal Irreducible Polynomials for $\mathbb{F}_{2^m}$ Arithmetic. *IACR Cryptology ePrint Archive*, page 192, 2007.

[59] Shanks Daniel. Class Number, a Theory of Factorization and Genera. In *Proceedings of Symposia in Pure Mathematics*, volume 20, pages 415–440. American Mathematical Society, 1971.

[60] Jerome A. Solinas. Generalized Mersenne Numbers. Technical report, Centre for Applied Cryptographic Research, University of Waterloo, 1999. http://cacr.uwaterloo.ca/techreports/1999/corr99-39.ps.

[61] Douglas Stinson. *Cryptographie, théorie et pratique*. Vuibert, 2003.

[62] Taher El Gamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In *Advances in Cryptology, Proceedings of CRYPTO, Proceedings*, pages 10–18, 1984.

[63] Andrew S. Tannenbaum. Modern Operating Systems, 2009. http://www.freewebs.com/ictft/sisop/Tanenbaum_Chapter2.pdf.

[64] Jonathan Taverne, Armando Faz-Hernández, Diego F. Aranha, Francisco Rodríguez-Henríquez, Darrel Hankerson, and Julio López. Software Implementation of Binary Elliptic Curves : Impact of the Carry-Less Multiplier on Scalar Multiplication. In *Cryptographic Hardware and Embedded Systems - CHES*, volume 6917 of *LNCS*, pages 108–123. Springer, 2011.

[65] Jonathan Taverne, Armando Faz-Hernández, Diego F. Aranha, Francisco Rodríguez-Henríquez, Darrel Hankerson, and Julio López. Speeding Scalar Multiplication over Binary Elliptic Curves using the New Carry-less Multiplication Instruction. *J. Cryptographic Engineering*, 1(3) :187–199, 2011.

[66] Ingrid Verbauwhede, Dusko Karaklajic, and Jörn-Marc Schmidt. The Fault Attack Jungle - A Classification Model to Guide You. In *FDTC*, pages 3–8, 2011.

[67] Yongbin Zhou and Dengguo Feng. Side-Channel Attacks : Ten Years After Its Publication and the Impacts on Cryptographic Module Security Testing. *IACR Cryptology ePrint Archive*, page 388, 2005.

Résumé

Avec le développement des communications et de l'Internet, l'échange des informations cryptées a explosé. Cette évolution a été possible par le développement des protocoles de la cryptographie asymétrique qui font appel à des opérations arithmétiques telles que l'exponentiation modulaire sur des grands entiers ou la multiplication scalaire de point de courbe elliptique. Ces calculs sont réalisés par des plates-formes diverses, depuis la carte à puce jusqu'aux serveurs les plus puissants. Ces plates-formes font l'objet d'attaques qui exploitent les informations recueillies par un canal auxiliaire, tels que le courant instantané consommé ou le rayonnement électromagnétique émis par la plate-forme en fonctionnement.

Dans la thèse, nous améliorons les performances des opérations résistantes à l'attaque *Simple Power Analysis*. Sur l'exponentiation modulaire, nous proposons d'améliorer les performances par l'utilisation de multiplications modulaires multiples avec un opérande commun optimisées. Nous avons proposé trois améliorations sur la multiplication scalaire de point de courbe elliptique : sur corps binaire, nous employons des améliorations sur les opérations combinées AB, AC et $AB + CD$ sur les approches *Double-and-add*, *Halve-and-add* et *Double/halve-and-add* et l'échelle binaire de Montgomery ; sur corps binaire, nous proposons de paralléliser l'échelle binaire de Montgomery ; nous réalisons l'implantation d'une approche parallèle de l'approche *Right-to-left Double-and-add* sur corps premier et binaire, *Halve-and-add* et *Double/halve-and-add* sur corps binaire.

Abstract

The development of online communications and the Internet have made encrypted data exchange fast growing. This has been possible with the development of asymmetric cryptographic protocols, which make use of arithmetic computations such as modular exponentiation of large integer or elliptic curve scalar multiplication. These computations are performed by various platforms, including smart-cards as well as large and powerful servers. The platforms are subjects to attacks taking advantage of information leaked through side channels, such as instantaneous power consumption or electromagnetic radiations.

In this thesis, we improve the performance of cryptographic computations resistant to *Simple Power Analysis*. On modular exponentiation, we propose to use multiple multiplications sharing a common operand to achieve this goal. On elliptic curve scalar multiplication, we suggest three different improvements : over binary fields, we make use of improved combined operations AB, AC and $AB + CD$ applied to *Double-and-add*, *Halve-and-add* and *Double/halve-and-add* approaches, and to the Montgomery ladder ; over binary field, we propose a parallel Montgomery ladder ; we make an implementation of a parallel approach based on the *Right-to-left Double-and-add* algorithm over binary and prime fields, and extend this implementation to the *Halve-and-add* and *Double/halve-and-add* over binary fields.